10-03-00

A

Please type a plus sign (+) inside this box → +

PTO/SB/05 (4/98)
Approved for use through 09/30/2000. OMB 0651-0032
Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

UTILITY PATENT APPLICATION TRANSMITTAL

(Only for new nonprovisional applications under 37 C.F.R. § 1.53(b))

Attorney Docket No.	155607-0086 (P002XD)
First Inventor or Application Identifier	Leonard J. Galasso
Title	System and Method for Securely Utilizing Basic Input and..
Express Mail Label No.	EL603887192US

APPLICATION ELEMENTS

See MPEP chapter 600 concerning utility patent application contents.

ADDRESS TO: Assistant Commissioner for Patents, Box Patent Application, Washington, DC 20231

1. [x] * Fee Transmittal Form (e.g., PTO/SB/17) (Submit an original and a duplicate for fee processing)
2. [x] Specification [Total Pages 68] (preferred arrangement set forth below)
   - Descriptive title of the Invention
   - Cross References to Related Applications
   - Statement Regarding Fed sponsored R & D
   - Reference to Microfiche Appendix
   - Background of the Invention
   - Brief Summary of the Invention
   - Brief Description of the Drawings (if filed)
   - Detailed Description
   - Claim(s)
   - Abstract of the Disclosure
3. [x] Drawing(s) (35 U.S.C. 113) [Total Sheets 19]
4. Oath or Declaration [Total Pages 4]
   a. [ ] Newly executed (original or copy)
   b. [x] Copy from a prior application (37 C.F.R. § 1.63(d)) (for continuation/divisional with Box 16 completed)
      i. [ ] DELETION OF INVENTOR(S) Signed statement attached deleting inventor(s) named in the prior application, see 37 C.F.R. §§ 1.63(d)(2) and 1.33(b).

* NOTE FOR ITEMS 1 & 13: IN ORDER TO BE ENTITLED TO PAY SMALL ENTITY FEES, A SMALL ENTITY STATEMENT IS REQUIRED (37 C.F.R. § 1.27), EXCEPT IF ONE FILED IN A PRIOR APPLICATION IS RELIED UPON (37 C.F.R. § 1.28).

5. [ ] Microfiche Computer Program (Appendix)
6. Nucleotide and/or Amino Acid Sequence Submission (if applicable, all necessary)
   a. [ ] Computer Readable Copy
   b. [ ] Paper Copy (identical to computer copy)
   c. [ ] Statement verifying identity of above copies

ACCOMPANYING APPLICATION PARTS

7. [x] Assignment Papers (cover sheet & document(s))
8. [x] 37 C.F.R.§3.73(b) Statement (when there is an assignee) [x] Power of Attorney
9. [ ] English Translation Document (if applicable)
10. [x] Information Disclosure Statement (IDS)/PTO-1449 [x] Copies of IDS Citations
11. [x] Preliminary Amendment
12. [x] Return Receipt Postcard (MPEP 503) (Should be specifically itemized)
13. [ ] * Small Entity Statement(s) (PTO/SB/09-12) [ ] Statement filed in prior application, Status still proper and desired
14. [ ] Certified Copy of Priority Document(s) (if foreign priority is claimed)
15. [x] Other: Revocation & Power of Attorney (2 pgs.)and Appendices (59 pgs.) copied from a prior appl.

16. If a CONTINUING APPLICATION, check appropriate box, and supply the requisite information below and in a preliminary amendment:

[ ] Continuation [x] Divisional [ ] Continuation-in-part (CIP) of prior application No: 09 / 336,889

Prior application information: Examiner Nguyen, T. Group / Art Unit: 2759

For CONTINUATION or DIVISIONAL APPS only: The entire disclosure of the prior application, from which an oath or declaration is supplied under Box 4b, is considered a part of the disclosure of the accompanying continuation or divisional application and is hereby incorporated by reference. The incorporation can only be relied upon when a portion has been inadvertently omitted from the submitted application parts.

17. CORRESPONDENCE ADDRESS

[ ] Customer Number or Bar Code Label (Insert Customer No. or Attach bar code label here) or [x] Correspondence address below

Name	Kimberley G. Nobles				
	IRELL & MANELLA LLP				
Address	840 Newport Center Drive				
	Suite 400				
City	Newport Beach	State	CA	Zip Code	92660
Country	USA	Telephone	(949) 760-0991	Fax	(949) 760-5200

Name (Print/Type)	Babak Redjaian	Registration No. (Attorney/Agent)	42,096
Signature	Babak Redj	Date	October 3, 2000

JC925 U.S. PTO
09/679450
10/03/00

PTO/SB/17 (12/99)
Approved for use through 09/30/2000. OMB 0651-0032
Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

FEE TRANSMITTAL for FY 2000

Patent fees are subject to annual revision.
Small Entity payments must be supported by a small entity statement, otherwise large entity fees must be paid. See Forms PTO/SB/09-12. See 37 C.F.R. §§ 1.27 and 1.28.

TOTAL AMOUNT OF PAYMENT ($) $768.00

Complete if Known	
Application Number	NEW
Filing Date	
First Named Inventor	Leonard J. Galasso
Examiner Name	
Group / Art Unit	
Attorney Docket No.	155607-0086 (P002XD)

METHOD OF PAYMENT (check one)

1. ☑ The Commissioner is hereby authorized to charge indicated fees and credit any overpayments to:

Deposit Account Number: 09-0946

Deposit Account Name: Irell & Manella LLP

☑ Charge Any Additional Fee Required Under 37 CFR §§ 1.16 and 1.17

2. ☑ **Payment Enclosed:** ☑ Check ☐ Money Order ☐ Other

FEE CALCULATION

1. BASIC FILING FEE

Large Entity Fee Code	Large Entity Fee ($)	Small Entity Fee Code	Small Entity Fee ($)	Fee Description	Fee Paid
101	690	201	345	Utility filing fee	690
106	310	206	155	Design filing fee	
107	480	207	240	Plant filing fee	
108	690	208	345	Reissue filing fee	
114	150	214	75	Provisional filing fee	

SUBTOTAL (1) ($) 690.00

2. EXTRA CLAIM FEES

			Extra Claims		Fee from below		Fee Paid
Total Claims	16	-20** =	0	X	18	=	0
Independent Claims	4	- 3** =	1	X	78	=	78
Multiple Dependent						=	

***or number previously paid, if greater; For Reissues, see below*

Large Entity Fee Code	Large Entity Fee ($)	Small Entity Fee Code	Small Entity Fee ($)	Fee Description
103	18	203	9	Claims in excess of 20
102	78	202	39	Independent claims in excess of 3
104	260	204	130	Multiple dependent claim, if not paid
109	78	209	39	** Reissue independent claims over original patent
110	18	210	9	** Reissue claims in excess of 20 and over original patent

SUBTOTAL (2) ($) 78.00

FEE CALCULATION (continued)

3. ADDITIONAL FEES

Large Entity Fee Code	Large Entity Fee ($)	Small Entity Fee Code	Small Entity Fee ($)	Fee Description	Fee Paid
105	130	205	65	Surcharge - late filing fee or oath	
127	50	227	25	Surcharge - late provisional filing fee or cover sheet.	
139	130	139	130	Non-English specification	
147	2,520	147	2,520	For filing a request for reexamination	
112	920*	112	920*	Requesting publication of SIR prior to Examiner action	
113	1,840*	113	1,840*	Requesting publication of SIR after Examiner action	
115	110	215	55	Extension for reply within first month	
116	380	216	190	Extension for reply within second month	
117	870	217	435	Extension for reply within third month	
118	1,360	218	680	Extension for reply within fourth month	
128	1,850	228	925	Extension for reply within fifth month	
119	300	219	150	Notice of Appeal	
120	300	220	150	Filing a brief in support of an appeal	
121	260	221	130	Request for oral hearing	
138	1,510	138	1,510	Petition to institute a public use proceeding	
140	110	240	55	Petition to revive - unavoidable	
141	1,210	241	605	Petition to revive - unintentional	
142	1,210	242	605	Utility issue fee (or reissue)	
143	430	243	215	Design issue fee	
144	580	244	290	Plant issue fee	
122	130	122	130	Petitions to the Commissioner	
123	50	123	50	Petitions related to provisional applications	
126	240	126	240	Submission of Information Disclosure Stmt	
581	40	581	40	Recording each patent assignment per property (times number of properties)	
146	690	246	345	Filing a submission after final rejection (37 CFR § 1.129(a))	
149	690	249	345	For each additional invention to be examined (37 CFR § 1.129(b))	

Other fee (specify) ____________

Other fee (specify) ____________

* Reduced by Basic Filing Fee Paid

SUBTOTAL (3) ($)

SUBMITTED BY			Complete (if applicable)		
Name (Print/Type)	Babak Redjaian	Registration No. (Attorney/Agent)	42,096	Telephone	(949) 760-0991
Signature	Babak Redjaian			Date	10/3/00

WARNING:

Information on this form may become public. Credit card information should not be included on this form. Provide credit card information and authorization on PTO-2038.

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of:

LEONARD J. GALASSO et al.

Application No.: Not Assigned

Filed:

For: SYSTEM AND METHOD FOR SECURELY UTILIZING BASIC INPUT AND OUTPUT SYSTEM (BIOS) SERVICES

Examiner: Not Assigned

Art Group: Not Assigned

Which is a Divisional Application of:

LEONARD J. GALASSO et al.

Application No.: 09/336,889

Filed: June 18, 1999

For: SYSTEM AND METHOD FOR SECURELY UTILIZING BASIC INPUT AND OUTPUT SYSTEM (BIOS) SERVICES

Examiner: Nguyen, T.

Art Group: 2759

PRELIMINARY AMENDMENT

ASSISTANT COMMISSIONER
FOR PATENTS
Washington, D.C. 20231

Dear Sir:

Prior to examination of the above-identified application, please enter the Preliminary Amendment as follows.

REMARKS

Please cancel claims 1 through 36. Consequently, claims 37 through 52 remain pending in this application for examination on the merits. Applicant respectfully requests examination of

the pending claims at the Examiner's earliest convenience. Should there be any issues that may be resolved through a telephone interview, the Examiner is requested to telephone the undersigned.

Respectfully submitted,

IRELL & MANELLA LLP

Dated: October 3, 2000

Babak Redj

Babak Redjaian, Reg. No. 42,096

840 Newport Center Drive
Suite 400
Newport Beach, CA 92660
Telephone: (949) 760-0991

Certificate of Mailing

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail No. EL603887127US in an envelope addressed to: Assistant Commissioner For Patents, Washington, D.C. 20231 on October 3, 2000.

Darla Cleveland — 10/3/00
Date

Docket No.: 002603.P002X
Express Mail No. EL236788568

UNITED STATES PATENT APPLICATION

FOR

SYSTEM AND METHOD FOR SECURELY UTILIZING BASIC INPUT AND OUTPUT SYSTEM (BIOS) SERVICES

Inventors: Leonard J. Galasso
Matthew E. Zilmer
Quang Phan

Prepared By:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 Wilshire Blvd., 7th Floor
Los Angeles, California 90025-1026
(310) 207-3800

RELATED APPLICATION

This application is a Continuation-In-Part of U.S. Patent Application No. 08/947,990 filed on October 9, 1997.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a system and method for securely utilizing Basic Input and Output System (BIOS) services.

2. Description of the Related Art

In virtual memory subsystems, "virtual" memory addressing is employed in which the memory addresses utilized in software programs are mapped indirectly to locations in physical memory. Translation to physical addresses is typically accomplished by the processor, and such physical addresses are inaccessible to user mode software and the Basic Input/Output System (BIOS).

One example of such virtual memory subsystems is that used by Windows NT, which is manufactured and marketed by Microsoft, Inc. In particular, Windows NT incorporates a demand-paged virtual memory subsystem. The memory address space provided to a program running on the Windows NT operating system is safeguarded from other user mode programs just as other programs are protected from it. This ensures that user mode services and applications will not write over

each other's memory, or execute each other's instructions. Kernel mode services and applications are protected in a similar way. If an attempt to access memory outside of a program's allocated virtual space occurs, the program is terminated and the user is notified. Virtual memory subsystems also prevent direct access by user mode software to physical memory addresses and to input/output devices that are part of a computer system.

There is an increasing trend towards the use of input/output devices on a computer system which are capable of executing operating systems using virtual memory subsystems. In such systems, there is no means for accessing memory outside of a program's virtual memory space, such as BIOS functions. One approach to this problem is to install a device driver which reads a file containing instructions for a device. The driver reads the file and writes (or downloads) these instructions into the device's memory. However, this type of device driver permits only limited addressing capability for memory and input/output operations. In addition, it does not allow execution of the system's processor instructions in physical memory space.

Accordingly, there is a need in the technology for a system and method for accessing and executing the contents of physical memory from a virtual memory subsystem, which facilitates increased addressing capability for memory and input/output operations, and which also allows execution of processor instructions directly from physical memory.

Furthermore, data stored on computer systems or platforms can be updated or configured. In certain cases, the data is extremely sensitive. A good example of configurable sensitive data is the Basic Input and Output System (BIOS) of a

computer system. Typically stored in some form of non-volatile memory, the BIOS is machine code, usually part of an Operating System (OS), which allows the Central Processing Unit (CPU) to perform tasks such as initialization, diagnostics, loading the operating system kernel from mass storage, and routine input/output ("I/O") functions. Upon power up, the CPU will "boot up" by fetching the instruction code residing in the BIOS. Without any security protection, the BIOS is vulnerable to attacks through capturing and replaying of service requests to invoke functions provided by the BIOS. These attacks may corrupt the BIOS and disable the computer system.

Accordingly, there is also need to provide a system and method to verify the integrity of service requests to access or modify data in the BIOS and to enforce proper authorization limits of those remote request messages.

SUMMARY OF THE INVENTION

The present invention provides a system and method for securely utilizing Basic Input and Output System (BIOS) services.

In accordance with one aspect of the current invention, the system comprises a memory for storing instruction sequences by which the processor-based system is processed, where the memory includes a physical memory and a virtual memory. The system also comprises a processor for executing the stored instruction sequences. The stored instruction sequences include process acts to cause the processor to: map a plurality of predetermined instruction sequences from the physical memory to the virtual memory, determine an offset to one of the plurality of predetermined instruction sequences in the virtual memory, receive an instruction to execute the one of the plurality of predetermined instruction sequences, transfer control to the one of the plurality of predetermined instruction sequences, and process the one of the plurality of predetermined instruction sequences from the virtual memory.

Another aspect of the system includes an access driver to generate a service request to utilize BIOS services such that the service request contains a service request signature created using a private key in a cryptographic key pair. The system also includes an interface to verify the service request signature using a public key in the cryptographic key pair to ensure integrity of the service request.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a system block diagram of one embodiment of an information distribution system in which the and method of the invention is used.

Figure 2 illustrates an exemplary processor system or user computer system which implements embodiments of the present invention.

Figure 3 illustrates a diagram of one embodiment of the computer system of Figure 2, in which the system and method of invention is used.

Figure 4 is an overall functional block diagram illustrating the architecture of an operating system which utilizes the system and method of the present invention.

Figure 5 is a block diagram illustrating the access driver 46 initialization process as provided in accordance with the principles of the present invention.

Figure 6A is a flowchart illustrating one embodiment of the initialization process of the present invention.

Figure 6B is a flowchart illustrating the details of process block 610 of Figure 6A.

Figure 6C is a flowchart illustrating the details of process block 630 of Figure 6A.

Figure 7A is a flowchart illustrating the execution process of the present invention.

Figure 7B is a flowchart illustrating the details of process block 640 of Figure 7A.

Figure 8 is another overall functional block diagram illustrating the architecture of an operating system which utilizes the system and method of the present invention.

Figure 9 shows an illustrative sequence of interactive sequence between two system components in accordance with one embodiment of the current invention.

Figure 10 outlines the generation of a session request in accordance with one embodiment of the current invention.

Figure 11 shows an authority certificate in accordance with one embodiment of the current invention.

Figure 12 shows a session request in accordance with one embodiment of the current invention.

Figure 13 shows a service request in accordance with one embodiment of the current invention.

Figure 14 outlines the acts required to establish a work session in accordance with one embodiment of the current invention.

Figure 15 outlines the generation of a service request in accordance with one embodiment of the current invention.

Figure 16 shows the acts required in processing of a service request in accordance with one embodiment of the current invention.

Figure 17 shows the acts involved in ending the current work session in accordance with one embodiment of the current invention.

Figure 18 shows the process of generating an authority certificate in accordance with one embodiment of the current invention.

DETAILED DESCRIPTION OF THE PREFERRED INVENTION

Definitions

As discussed herein, a "computer system" is a product including circuitry capable of processing data. The computer system may include, but is not limited to, general purpose computer systems (e.g., server, laptop, desktop, palmtop, personal electronic devices, etc.), personal computers (PCs), hard copy equipment (e.g., printer, plotter, fax machine, etc.), banking equipment (e.g., an automated teller machine), and the like. An infomediary is a web site that provides information on behalf of producers of goods and services, supplying relevant information to businesses about products and/or services offered by suppliers and other businesses. Content refers to application programs, driver programs, utility programs, the payload, etc., and combinations thereof, as well as graphics, informational material (such as articles, stock quotes, etc.) and the like, either singly or in any combination. In addition, a "communication link" refers to the medium or channel of communication. The communication link may include, but is not limited to, a telephone line, a modem connection, an Internet connection, an Integrated Services Digital Network ("ISDN") connection, an Asynchronous Transfer Mode (ATM) connection, a frame relay connection, an Ethernet connection, a coaxial connection, a fiber optic connection, satellite connections (e.g. Digital Satellite Services, etc.), wireless connections, radio frequency (RF) links, electromagnetic links, two way paging connections, etc., and combinations thereof. Power On Self Test (POST) refers to the instructions that are executed to configure and test the system hardware prior to loading an OS.

System Overview

A description of an exemplary system, which incorporates embodiments of the present invention, is hereinafter described.

Figure 1 shows a system block diagram of one embodiment of an information distribution system 10 in which the system and method of the invention is used. The system 10 relates to providing an infomediary. It involves the construction and maintenance of a secure and private repository of Internet user and system profiles, collected primarily from warranty service registrations, Internet service registrations, system profiles, and user preferences. Initially, this information is used to register the user with the manufacturers of purchased hardware and software products, and with the providers of on-line or other services. Over time, the user data is used to create a user profile and notify users of relevant software updates and upgrades, to encourage on-line purchases of related products, and to enable one-to-one customized marketing and other services.

In one embodiment, two software modules are used to implement various embodiments of the invention. One is resident on a user's system, and is used to access a predetermined web site. For example, in one embodiment, the operating system and Basic Input and Output System (BIOS) are pre-installed on a computer system, and when the computer system is subsequently first powered up, an application, referred to for discussion purposes as the first software module (in one embodiment, the first software module is the initial start-up application (ISUA), which will be described in the following sections), will allow the launching of one or more executable programs in the preboot environment. In one embodiment, the first software module facilitates the launching of one or more executable programs prior to the loading, booting, execution and/or running of the OS. In one

embodiment, the user is encouraged to select the use of such a program (i.e., the use of the first software module), and in alternative embodiments, the program is automatically launched. The program(s) contained in the first software module enables tools and utilities to run at an appropriate time, and with proper user authorization, also allow the user to download a second software module that includes drivers, applications and additional payloads through the Internet connection on the PC. The programs may also provide for remote management of the system if the OS fails to launch successfully.

Once the second software module has been delivered, it may become memory resident, and may disable the transferred copy of the first software module. The original copy of the first software module still residing in the system's non-volatile memory remains idle until the second software module fails to function, becomes corrupted or is deleted, upon which a copy of the original first software module is again transferred as described above. The second software module may include an application that connects the user to a specific server on the Internet and directs the user to a predetermined web site to seek authorization to down load further subscription material. The second software module may also include content that is the same or similar to the content of the first software module.

In one embodiment, the system may also include an initial payload that is stored in Read Only Memory BIOS (ROM BIOS). In one embodiment, the initial payload is part of the first software module (e.g., the ISUA). In an alternative embodiment, the initial payload is stored as a module in ROM BIOS, separate from the first software module. In one embodiment, the initial payload is launched from ROM BIOS and displayed on the screen after the Power On Self Test (POST) but prior to the booting, loading and/or execution of the OS. This may occur at a predetermined time, such as when the system is being manufactured, assembled

and tested, or when the end user first activates the system. In an alternate embodiment, this initial payload is copied to a predetermined location (such as the system's hard disk) at a predetermined time, such as when the system is being manufactured, assembled and tested, or when the end user first activates the system. Once copied, the payload executes after POST but prior to operation of the OS, and may display graphics, advertisements, animation, Joint Photographic Experts Group (JPEG)/Moving Picture Experts Group (MPEG) formatted material on the screen. When additional programs and/or payloads are delivered (via the Internet or other outside connection), the display screen may be used to provide customized screens in the form of messages or graphics prior to and during booting of the OS. In addition, executable programs delivered in the first software module, as well as subsequent programs (such as the second software module) downloaded from the web site, may be used to survey the PC to determine various types of devices, drivers, and applications installed. In one embodiment, as described in co-pending U.S. Patent Application serial number ___, entitled "Method and Apparatus for Automatically Installing And Configuring Software on a Computer", filed June 18, 1999, assigned to Phoenix Technologies Ltd., the contents of which are incorporated herein by reference, the first software module is used to identify and to automatically create shortcuts and/or bookmarks for the user. The programs downloaded from the website may include software that collects and maintains a user profile based on the user's preferences. Such information may be provided to the infomediary, which subsequently forwards portions of the information and/or compiled data based on the information to suppliers and other businesses to obtain updates or revisions of information provided by the suppliers and other businesses.

Referring to Figure 1, the information distribution system 10 comprises a service center 20 that is connected over one or more communications links 30_1-30_N

to one or more user computer systems 40_1–40_N ("40"). The service center 20 includes one or more servers 22, one or more databases 24, and one or more computers 26_1-26_M. The one or more computers 26_1-26_M are capable of simultaneous access by a plurality of the user computer systems 40_1-40_N. If a plurality of computers are used, then the computers 26_1-26_M may be connected by a local area network (LAN) or any other similar connection technology. However, it is also possible for the service center 20 to have other configurations. For example, a smaller number of larger computers (i.e. a few mainframe, mini, etc. computers) with a number of internal programs or processes running on the larger computers capable of establishing communications links to the user computers.

The service center 20 may also be connected to a remote network 50 (e.g., the Internet) or a remote site (e.g., a satellite, which is not shown in Figure 1). The remote network 50 or remote site allows the service center 20 to provide a wider variety of computer software, content, etc. that could be stored at the service center 20. The one or more databases 24 connected to the service center computer(s), e.g., computer 26_1, are used to store database entries consisting of computer software available on the computer(s) 26. In one embodiment, each user computer 40_1-40_N has its own secure database (not shown), that is not accessible by any other computer. The communication links 30_1-30_N allow the one or more user computer systems 40_1-40_N to simultaneously connect to the computer(s) 26_1-26_M. The connections are managed by the server 22.

After a user computer system 40 establishes two-way communications with the information service computer 26, the content is sent to the user computer system 40 in a manner hereinafter described. The downloaded content includes an application that surveys the user and/or the user computer system's hardware and/or software to develop a user profile as well as a profile of the user's system.

The information gathered from the user and/or user's computer system is subsequently provided to the service center 20, which provides additional content to the user computer 40 based on the user and system profile. The database entries from the database connected to the service computer 26 contain information about computer software, hardware, and third party services and products that are available to a user. Based on the user and/or system profile, the content is further sent to the user computer for display. The content may also include a summary of information such as the availability of patches and fixes for existing computer software, new versions of existing computer software, brand new computer software, new help files, etc. The content may further include information regarding availability of hardware and third party products and services that is of interest to the user. The user is then able to make one or more choices from the summary of available products and services, and request that the products be transferred from the service computer 26 to the user computer. Alternatively, the user may purchase the desired product or service from the summary of available products and services.

Figure 2 illustrates an exemplary computer system 100 that implements embodiments of the present invention. The computer system 100 illustrates one embodiment of user computer systems 40_1-40_N and/or computers 26_1-26_M (Figure 1), although other embodiments may be readily used.

Referring to Figure 2, the computer system 100 comprises a processor or a central processing unit (CPU) 104. The illustrated CPU 104 includes an Arithmetic Logic Unit (ALU) for performing computations, a collection of registers for temporary storage of data and instructions, and a control unit for controlling operation for the system 100. In one embodiment, the CPU 104 includes any one of the x86, Pentium™, Pentium II™, and Pentium Pro™ microprocessors as marketed

by Intel™ Corporation, the K-6 microprocessor as marketed by AMD™, or the 6x86MX microprocessor as marketed by Cyrix™ Corp. Further examples include the Alpha™ processor as marketed by Digital Equipment Corporation™, the 680X0 processor as marketed by Motorola™; or the Power PC™ processor as marketed by IBM™. In addition, any of a variety of other processors, including those from Sun Microsystems, MIPS, IBM, Motorola, NEC, Cyrix, AMD, Nexgen and others may be used for implementing CPU 104. The CPU 104 is not limited to microprocessor but may take on other forms such as microcontrollers, digital signal processors, reduced instruction set computers (RISC), application specific integrated circuits, and the like. Although shown with one CPU 104, computer system 100 may alternatively include multiple processing units.

The CPU 104 is coupled to a bus controller 112 by way of a CPU bus 108. The bus controller 112 includes a memory controller 116 integrated therein, though the memory controller 116 may be external to the bus controller 112. The memory controller 116 provides an interface for access by the CPU 104 or other devices to system memory 124 via memory bus 120. In one embodiment, the system memory 124 includes synchronous dynamic random access memory (SDRAM). System memory 124 may optionally include any additional or alternative high speed memory device or memory circuitry. The bus controller 112 is coupled to a system bus 128 that may be a peripheral component interconnect (PCI) bus, Industry Standard Architecture (ISA) bus, etc. Coupled to the system bus 128 are a graphics controller, a graphics engine or a video controller 132, a mass storage device 152, a communication interface device 156, one or more input/output (I/O) devices 168_1-168_N, and an expansion bus controller 172. The video controller 132 is coupled to a video memory 136 (e.g., 8 Megabytes) and video BIOS 140, all of which may be integrated onto a single card or device, as designated by numeral 144. The video

memory 136 is used to contain display data for displaying information on the display screen 148, and the video BIOS 140 includes code and video services for controlling the video controller 132. In another embodiment, the video controller 132 is coupled to the CPU 104 through an Advanced Graphics Port (AGP) bus.

The mass storage device 152 includes (but is not limited to) a hard disk, floppy disk, CD-ROM, DVD-ROM, tape, high density floppy, high capacity removable media, low capacity removable media, solid state memory device, etc., and combinations thereof. The mass storage device 152 may include any other mass storage medium. The communication interface device 156 includes a network card, a modem interface, etc. for accessing network 164 via communications link 160. The I/O devices 168_1-168_N include a keyboard, mouse, audio/sound card, printer, and the like. The I/O devices 168_1-168_N may be a disk drive, such as a compact disk drive, a digital disk drive, a tape drive, a zip drive, a jazz drive, a digital video disk (DVD) drive, a solid state memory device, a magneto-optical disk drive, a high density floppy drive, a high capacity removable media drive, a low capacity media device, and/or any combination thereof. The expansion bus controller 172 is coupled to nonvolatile memory 175 which includes system firmware 176. The system firmware 176 includes system BIOS 82, which is for controlling, among other things, hardware devices in the computer system 100. The system firmware 176 also includes ROM 180 and flash (or EEPROM) 184. The expansion bus controller 172 is also coupled to expansion memory 188 having RAM, ROM, and/or flash memory (not shown). The system 100 may additionally include a memory module 190 that is coupled to the bus controller 112. In one embodiment, the memory module 190 comprises a ROM 192 and flash (or EEPROM) 194.

As is familiar to those skilled in the art, the computer system 100 further includes an operating system (OS) and at least one application program, which in

one embodiment, are loaded into system memory 124 from mass storage device 152 and launched after POST. The OS may include any type of OS including, but not limited or restricted to, DOS, Windows™ (e.g., Windows 95™, Windows 98™, Windows NT™), Unix, Linux, OS/2, OS/9, Xenix, etc. The operating system is a set of one or more programs which control the computer system's operation and the allocation of resources. The application program is a set of one or more software programs that performs a task desired by the user.

In accordance with the practices of persons skilled in the art of computer programming, the present invention is described below with reference to symbolic representations of operations that are performed by computer system 100, unless indicated otherwise. Such operations are sometimes referred to as being computer-executed. It will be appreciated that operations that are symbolically represented include the manipulation by CPU 104 of electrical signals representing data bits and the maintenance of data bits at memory locations in system memory 124, as well as other processing of signals. The memory locations where data bits are maintained are physical locations that have particular electrical, magnetic, optical, or organic properties corresponding to the data bits.

When implemented in software, the elements of the present invention are essentially the code segments to perform the necessary tasks. The program or code segments can be stored in a processor readable medium or transmitted by a computer data signal embodied in a carrier wave over a transmission medium or communication link. The "processor readable medium" may include any medium that can store or transfer information. Examples of the processor readable medium include an electronic circuit, a semiconductor memory device, a ROM, a flash memory, an erasable ROM (EROM), a floppy diskette, a CD-ROM, an optical disk, a hard disk, a fiber optic medium, a radio frequency (RF) link, etc. The computer data

signal may include any signal that can propagate over a transmission medium such as electronic network channels, optical fibers, air, electromagnetic, RF links, etc. The code segments may be downloaded via computer networks such as the Internet, Intranet, etc.

Figure 3 illustrates a logical diagram of computer system 100. Referring to Figures 2 and 3, the system firmware 176 includes software modules and data that are loaded into system memory 124 during POST and subsequently executed by the processor 104. In one embodiment, the system firmware 176 includes a system BIOS module 82 having system BIOS handlers, hardware routines, etc., a ROM application program interface (RAPI) module 84, an initial start-up application (ISUA) module 86, an initial payload 88a, cryptographic keys 90, a cryptographic engine 92, and a display engine 94. The aforementioned modules and portions of system firmware 176 may be contained in ROM 180 and/or flash 184. Alternatively, the aforementioned modules and portions of system firmware 176 may be contained in ROM 190 and/or flash 194. The RAPI 84, ISUA 86, and initial payload 88a may each be separately developed and stored in the system firmware 176 prior to initial use of the computer system 100. In one embodiment, the RAPI 84, ISUA 86, and initial payload 88a each includes proprietary software developed by Phoenix Technologies, Ltd.

RAPI 84 generally provides a secured interface between ROM application programs and system BIOS 82. One embodiment of RAPI 84 is described below in Figures 8 through 18 and the accompanying text. One embodiment of ISUA 86 is described in co-pending U.S. Patent Application serial number ___ entitled "Method and Apparatus for Automatically Installing and Configuring Software on a Computer," filed on June 18, 1999, assigned to Phoenix Technologies, Ltd., and which is incorporated herein by reference.

Accessing and Executing Contents of Physical Memory From Virtual Memory

One aspect of the present invention is described with reference to an operating system installed on the processing system 100, shown in Figure 2. Figure 4 is an overall functional block diagram illustrating the architecture of a processing system utilizing the system and method of the present invention. The processing system 100 comprises an operating system 2230 which supports application programs 232 and services 2234, Basic Input/Output System ("BIOS") 236 and system hardware 238. The BIOS 236 is a collection of drivers, or software interfaces for hardware devices such as the console (keyboard and display), a generic printer, the auxiliary device (serial port), the computer's clock and the boot disk device. The BIOS 236 is typically embedded in programmable, read only memory (ROM). Often, the BIOS functions themselves are actually copied from ROM into physical memory, taking advantage of the faster access times of physical memory. This is known as "shadowing" the BIOS 236 because two copies of BIOS 236 results: one in ROM (which will no longer be used) and the other in physical mempry. The portion of physical memory which stores the BIOS 236 is known as the BIOS shadow space. An operating system such as Windows NT makes no use of the BIOS 236 after the operating system has been booted and is running. The kernel level drivers in the Windows NT operating system interface directly with the system hardware. The present invention facilitates the use of the BIOS 236 as an interface between system hardware 238 and an operating system 232.

The operating system 230 includes a class driver 240 which interfaces with the application programs 232 and services 2234, and an I/O Manager 242. The I/O Manager 242 converts I/O requests from the application programs 232 and services 234 (made via the class driver 240) into properly sequenced calls to various driver

routines located in the kernel 244. In particular, when the I/O Manager 242 receives an I/O request, it uses the function codes of the request to call one of several dispatch routines in a driver located in the kernel 244. The kernel 244 provides hardware-independent functions, called system functions, that are accessed by means of a software interrupt. The functions provided by the kernel 244 include file and directory management, memory management, character device input/output and time and date support, among others. In one embodiment, the operating system is the Windows NT operating system. In alternate embodiments, the operating system 230 includes the Solaris or the AIX operating systems or other operating systems based on demand-paged virtual memory subsystems.

The present invention provides an access driver 246, located within the kernel 244, which is responsible for accessing BIOS data located in the BIOS 236 or for accessing system hardware 238 data via the BIOS 236. The access driver 246 is also responsible for accessing the location of a BIOS function address, as well as executing the associated BIOS function. In one preferred embodiment, the access driver 244 comprises source code written in the C language. It is understood that other assembly languages may be utilized in implementing the functions of the access driver 244. The BIOS data and addresses are typically located in physical memory 250 and are accessed by the access driver 246 via a BIOS Interface 248. In one embodiment, the access driver 246 executes code in the BIOS shadow space, typically at physical addresses 0x000E0000 through 0x000FFFFF.

By way of example, if the access driver 246 needs to access BIOS functions located in physical memory at address 0x00000000. It makes a call to the I/O Manager 242, requesting it to map the memory space at physical address 0x00000000 through 0x00000FFF to its virtual memory space. The I/O Manager 242 then returns a pointer to the virtual memory space of the access driver 246, for example,

0xfd268000. The access driver may now reference the address space at physical address 0x00000000 by basing or referring its virtual addresses with 0xfd268000. Thus, to access a function located at physical address 0x2400, the virtual address used would be 0xfd2682400.

In one preferred embodiment, a set of entry-points or functions calls are available to the application programs 230, services 232 or class driver 240 which utilize the access driver 246. The access driver 246 can be opened, closed, and can receive input/output ("I/O") control codes ("IOCTLs") through these entry points. Table 1 illustrates the structure, entry points and applications for the access driver 246.

Figure 5 is a block diagram illustrating the access driver 246 initialization process as provided in accordance with the principles of the present invention. In general, when the access driver 246 is first loaded, its DriverEntry function (see Table 1) is executed. Although a number of other initializations occur in this function (such as the allocation of various resources or objects for normal operation of the access driver 246), two particularly essential initializations occur: (a) the BIOS shadow area 260 (which includes a BIOS Service Directory 62) located in physical memory 250 and (b) the BIOS data area 264, also located in physical memory 250, are both mapped into the virtual memory (shown in Figure 5 as the BIOS shadow area 270 (which includes the BIOS Service Directory 272 and the BIOS data area 74) of the access driver 246. As a result, application programs 232 or services 2234 may, through the class driver 240, access or execute BIOS functions using virtual addressing. It should be noted that execution of the BIOS function must occur from the access driver 246 because the physical address space of the BIOS 236 is mapped only to the access driver 246. In addition, the access driver 246 may be utilized through the implementation of a 32-bit BIOS Power Management Service Interface

as described in detail in Appendix A. It is apparent to one skilled in the art that the BIOS Power Management Service Interface may also be implemented for 64-bit, 128-bit and 256-bit configurations. The access driver 246 can also operate in 64-bit, 128-bit and 256-bit configurations.

In particular, the access driver 246, during initialization, locates the BIOS shadow area 260 and the BIOS data area 264 located in physical memory 250. The BIOS shadow area 260 and the BIOS data area 264 are mapped into the virtual address space of the access driver 246. Next, the access driver 246 performs a search for the BIOS Service Directory 272 header. Upon finding and validating the BIOS Service Directory 272, the access driver 246 obtains the virtual address of the BIOS Service Directory 272 header, which provides the offset of the BIOS Service Directory 272 header virtual address from the base virtual address of the BIOS shadow area 270.

In an alternate embodiment, the access driver, during initialization, locates the BIOS shadow area 260, the BIOS data area 264 and the BIOS ROM area located in physical memory 250. The BIOS shadow area 260, the BIOS data area 264 and the BIOS ROM area are mapped into the virtual address space of the access driver 246. Next, the access driver 246 performs a search for the BIOS Service Directory 272 header. Upon finding and validating the BIOS Service Directory 272, the access driver 246 obtains the virtual address of the BIOS Service Directory 272 header. In this alternate embodiment, the availability of the BIOS ROM area in the virtual memory space of the access driver 246 enables the access driver 246 to read and/or write data in flash ROM. As a result, the BIOS ROM can be reflashed or rewritten. In addition, outside application programs which interface with hardware can access the BIOS ROM area through software mechanisms such as that described in the PhoenixPhlash NT specification provided in Appendix B.

Later, calls to an execution function in the access driver 246 will utilize the base virtual address of the BIOS shadow area 270 and the offset to invoke a requested entry point in the BIOS itself. It should be noted that an application program 232 or service 234 may cause execution of the BIOS function anywhere in the BIOS' virtual address space, and not only through the BIOS Service Directory 272.

In one embodiment, the execution function that is called to invoke a requested entry point in the BIOS is the IOCTL_BIOS_EXEC function, which is described in Table 1. The IOCTL_BIOS_EXEC function creates a register stack in a buffer (which is specified by the calling application program 232 or service 234) located in main memory or DRAM. The contents of the stack are the desired register values at the time the BIOS function is invoked. The access driver 246 passes the register stack from the calling application program 232 or service 234. The procedure call itself is performed using a pointer to the function specified in the BIOS Service Directory 272. In one embodiment, the BIOS function called by IOCTL_BIOS_EXEC accepts a 4-byte signature as an argument and locates the BIOS function associated with the signature. Values passed back to the calling application program 232 or service 234 include the base virtual address of the BIOS function, and the offset from the base address of the service's entry point.

A general discussion of the structure, entry functions and applications for access driver 246 will now be provided.

Table 1. Access Driver 246 Functions

Function Name	Description	Remarks
DriverEntry	Kernel calls this function at load time to cause the driver to initialize itself.	

Access Driver CreateClose	Kernel calls this function to create or close out the driver's functionality.	
Access Driver Unload	Called by Kernel for Service Control Manager to unload the driver. This function frees all resources established at load time.	
IOCTL_Locate	Locates the BIOS 232-bit entry point using a string search. The entry point virtual address, the BIOS base virtual address and the BIOS data area virtual address are returned to the caller.	

Table 1. Access Driver 246 Functions
(continued)

Function Name	Description	Remarks
IOCTL_BIOS_Read	Reads data from either the BIOS ROM, shadow or BIOS data area and returns it to the user's buffer.	
IOCTL_BIOS_Write	Writes data to either the BIOS ROM, shadow or BIOS data area from the user's buffer.	
IOCTL_BIOS_Exec	Passes execution to a BIOS entry point via the 232-bit entry point	
IOCTL_RTC_Read	Reads the real-time clock date and time and returns these to the calling function.	Format of returned data will match SYSTEMTIME template.
IOCTL_Version	Returns the version number of the driver to the caller	Includes a bitmap of currently implemented functions
Access Driver DriverReg	Registers the calling device by its object name and canonical name.	The list of registered devices will be used to resume and suspend.
Access Driver DriverReg	Deregisters the calling device.	The device must have registered already.
IOCTL_PM_Suspend	The function sends Suspend IRP's to all registered devices.	
IOCTL_PM_Resume	The function sends Resume IRP's to all registered devices.	

A. Detailed description of the Access Driver 246 functions

1. The "DriverEntry" function

This entry point causes the driver to initialize its variables, map in the BIOS shadow and data areas, and to allocate resources for its normal operation. As each resource or object is allocated, it is tabulated into the variable 'phResAndFlags'; this allows a single function ('freeResources') to free up resources used by the driver, no matter the reason for the driver being unloaded. The resources allocated or connected to are as follows:

a. Create the Device Object - establishes a device object and name with the system.

b. Initialize Error Logging - creates a link to the Event Log services.

c. Set up the major function entry points.

d. Create a Symbolic link - for use by Service or Application layers.

e. Map in the BIOS shadow - makes this memory space accessible in virtual memory to the driver.

f. Map in the BIOS ROM - makes the ROM area accessible in virtual memory address space.

g. Map in the BIOS Data - makes this memory space accessible in virtual memory to the driver.

h. Locate the BIOS 232-bit entry point for use by this driver.

In one embodiment, the device object name is 'Laptop', which is required in order to service the nexus functions required by the Microsoft OEM Adaptation Kit (OAK). The corresponding symbolic link name is 'PhoenixAD'.

2. AccessDriverCreateClose

This function is used to inform the driver 246 when an application program 232 or service 234 makes a request to the system for a device handle, or when it closes a handle already obtained. The Access Driver 246 responds to this dispatch entry point by completing the request successfully, but changing no other state variable of the driver 246.

3. AccessDriverUnload

This dispatch entry point is called by the kernel on behalf of the Service Control Manager (SCM) or other application when it is necessary to remove the driver from the system (device close from (SCM)). The result of this function call is that all resources tabulated in 'phResAndFlags' are freed to the system and the request is completed successfully.

4. AccessDriverReg

The Access Driver 246 driver has the function of performing "nexus processing" for the power management model provided as part of the OEM Adaptation Kit (OAK). This function is integral to the emulation of power management for OEM and standard devices having knowledge of and a requirement to use the OAK control methods. The AccessDriverReg function registers devices into a linked list. It also selectively "deregisters" devices on request. Typically OAK compliant device drivers will make the call for registration when their DriverEntry function is executed (when they are first loaded). And as part of the DriverUnload function, each registered device must make the call to remove itself from Access Driver 246's linked list of devices needing power management services.

5. IOCTL Functions

Every interface between the service or application layer and the BIOS is handled by an IOCTL function in the Access Driver 246 driver. Each IOCTL transfer is performed in Buffered Mode, so that the input data to the driver and its output data are transferred through a common system buffer. The pointer to this buffer space is given in the Input/Output (I/O) Request Packet as Irp>AssociatedIrp.SystemBuffer. Upon being given control, the IOCTL (within the driver) will get the system buffer address and use its contents to perform the request. The results of the IOCTL function's execution will be placed in the same system buffer as was used for input.

Each IOCTL that is implemented in the Access Driver 246 driver has a unique data format for IOCTL input data and for its output data. As the functions are

described below, their data buffer formats and descriptions of each field are given. Buffer offsets are given in bytes. The minimum buffer size given for each function is a recommended malloc() size to use for the application program's user buffers. System buffer sizes will automatically be derived from the user buffers.

6. <u>IOCTL Locate</u>

The IOCTL_Locate function is the first dispatch entry point to be called by the application program 232 or service 234 after the driver 246 initializes. The function returns the addresses of the BIOS232 Service Interface, the base address of the BIOS shadow area, and the base address of the BIOS Data Area, in flat-model virtual address format (232 bit addresses). Note that the BIOS232 Service Interface is the single entry point for all BIOS functions executed from the driver level or kernel threads (see Appendix A). The BIOS232 Service Interface is the single entry point for all BIOS functions executed from the driver level or kernel threads (see Appendix A). These address spaces are guaranteed to be accessible to this driver (only) during the time the Access Driver 246 driver is loaded.

Input data: None, do not rely on buffer contents

Output data:	Offset 0:	PUCHAR -	BIOS Service Directory Offset into Shadow.
	Offset 4:	PUCHAR -	BIOS Shadow Area Base Virtual Address.
	Offset 8:	PUCHAR -	BIOS Data Area Base Virtual Address.

Offset 8:	PUCHAR -	BIOS Data Area Base Virtual Address.
Offset 12:	PUCHAR -	BIOS ROM Area Base Virtual Address.

Min. Buffer Size: 16 bytes

7. IOCTL_BIOS_Read

The IOCTL BIOS_Read function is a general purpose reader of either the BIOS ROM, shadow area, or the data area.

Input Data:	Offset 0:	ULONG -	Mode Flags
	Bit 0:	1 =	shadow area, 0 = data area.
	Bit 2:	1 =	ROM area (overrides Bit 0)
	Offset 4:	PUCHAR -	Offset into the BIOS area to start the read
	Offset 8:	ULONG -	Length of the read in bytes.
Output Data:	Offset 0:	ULONG -	Length of the actual read
	Offset 4:	UCHAR array -	the actual data read

Min. Buffer Size: 16 bytes

Note: If a 'short read' occurs because the offset into the BIOS area specified an overlap with the end of the mapped BIOS memory, no error is returned. Instead the

'actual data read' field accurately indicates how much of the data is valid in the system buffer.

8. IOCTL_BIOS_Write

The IOCTL_BIOS_Write function is a general purpose writer of either the BIOS ROM, shadow, or the data area.

Input Data:	Offset 0: ULONG -		Mode Flags
	Bit 0:	1 =	shadow area, 0 = data area.
	Bit 2:	1 =	ROM area (overrides Bit 0)
	Offset 4: PUCHAR -		Offset into the BIOS area to start the write
	Offset 8: ULONG -		Length of the write in bytes.
Output Data:	Offset 0: ULONG -		Length of the actual write (zero, or request size)
	Offset 4: UCHAR array - the actual data read		

Min. Buffer Size: 16 bytes

Note: Short writes are not permitted due to the possibility of data corruption.

9. IOCTL_BIOS_Exec

The IOCTL_BIOS_Exec function is used to execute a BIOS function through the BIOS232 Service Interface. An activation record is passed *by value* in the system buffer. The AR determines the Base Architecture register contents upon invocation

of the entry point to the BIOS. Upon successful completion, the AR contains the Base Architecture context that would normally have been returned to the BIOS caller.

Input Data:	Offset 0: ULONG - Function entry point virtual address.	
	Offset 4: ULONG - Flags Register	
	Offset 8: ULONG - reserved	
	Offset 12: ULONG -	reserved
	Offset 16: ULONG -	reserved
	Offset 20: ULONG -	reserved
	Offset 24: ULONG -	reserved
	Offset 28: ULONG -	reserved
	Offset 232: ULONG -	reserved
	Offset 236: ULONG -	reserved
	Offset 240: ULONG -	reserved
	Offset 244: ULONG -	EBP Register
	Offset 48: ULONG -	EDI Register
	Offset 52: ULONG -	ESI Register
	Offset 56: ULONG -	EDX Register
	Offset 260: ULONG -	ECX Register
	Offset 64: ULONG -	EBX Register
	Offset 68: ULONG -	EAX Register

Output Data: Contents of the system buffer are identical in structure. Register contents may have been influenced by the BIOS function requested.

Min. Buffer Size: 80 bytes.

100. IOCTL_RTC_Read

The IOCTL_RTC_Read function is used to read the contents of the RTC registers in the CMOS RAM. The data from this atomic read is formatted similarly to the SYSTEMTIME structure and returned to the user in the System Buffer.

Input Data: None, do not rely on buffer contents

Output Data: <uses SYSTEMTIME template as shown below>

Offset 0:	WORD	current year
Offset 2:	WORD	current month (January=1)
Offset 4:	WORD	current day of week (Sunday = 0)
Offset 6:	WORD	current day of month (calendar)
Offset 8:	WORD	current hour
Offset 100:	WORD	current minute
Offset 12:	WORD	current second
Offset 14:	WORD	current millisecond

Min. Buffer Size: 32 bytes.

Note that the Year field in the RTC is 8 bits wide. The contents of the Year field in the RTC will be recalculated to a SYSTEMTIME.Year 16 bit field containing the entire value of the current year, AD. Examples: RTC=00, Year=1980; RTC=23, Year = 2003. Also note that Legacy RTC devices do not provide the millisecond field

in their register set. Because of this, the current millisecond field in the Output Data for this function will always be set to zero.

11. IOCTL_VERSION

The IOCTL_Version function returns to the caller the major, an minor version of the Access Driver 246 driver. In addition, the functions implemented by this version of the driver are enumerated in a bitmap. The purpose of the bitmap is for services or higher level drivers to evaluate whether or not this version of the driver can be used for their purposes (at installation time, typically).

Input Data: None, do not rely on buffer contents

Output Data: Offset 0: WORD Major Version (X.)

Offset 2: WORD Minor Version (.x)

Offset 4: ULONG Bitmap of implemented functions (see below)

Bit 31: 1 if IOCTL_Locate implemented

Bit 230: 1 if IOCTL_BIOS_Read implemented

Bit 29: 1 if IOCTL_BIOS_Write implemented

Bit 28: 1 if IOCTL_BIOS_Exec implemented

Bit 27: reserved

Bit 26: reserved

Bit 25: reserved

Bit 24: 1 if IOCTL_RTC_Read implemented

Bit 23: reserved

Bit 22: reserved for Phlash interlock

Bit 21: reserved for online setup (NVRAM writes)

Bit 20 - 0: reserved for future expansion

Bit 230: 1 if IOCTL_BIOS_Read implemented

Min. Buffer Size: 16 bytes

12. IOCTL_PM_Suspend

The IOCTL_PM_Suspend function causes IRP_MJ_PNP_POWER, IRP_MN_LT_SUSPEND IRP's to be sent to each device that has registered itself using the Access Driver DriverReg entry point.

Input Data: None, do not rely on buffer contents

Output Data: None, do not rely on buffer contents

13. IOCTL_PM_Resume

The IOCTL_PM_Resume function causes IRP_MJ_PNP_POWER, IRP_MN_LT_RESUME IRP's to be sent to each device that has registered itself using the Access Driver DriverReg entry point.

Input Data: None, do not rely on buffer contents

Output Data: None, do not rely on buffer contents

B. <u>Error Codes Returned by Access Driver 246</u>

The following table defines the error status returned when an IRP is unsuccessfully or only partially completed. Conditions of termination of the functions are given as well. This table is necessary because there is not necessarily a one-to-one correspondence between NTSTATUS values known by the operating system and those used by the Access Driver 246 device driver. In order to reverse translate the codes back into strings usable by an applications writer or an end-user, it is mandatory that only NTSTATUS error codes be used.

Table 2. NTSTATUS Codes returned from Access Driver 246

NTSTATUS	Function	Condition at Termination
STATUS_SUCCESS	All	Request succeeded
STATUS_SOME_NOT_MAPPED	DriverEntry	One or more of memory or I/O areas could not be mapped in. Driver load fails.
STATUS_NOT_IMPLEMENTED	All	Requested function is not implemented in this driver.
STATUS_ACCESS_DENIED	All IOCTL functions	Device is unusable by caller because another service has exclusive access or because the BIOS Service Directory signature is not present.
STATUS_MEDIA_CHECK	All IOCTL functions - not implemented -	Contents of BIOS ROM may have been changed. New call to IOCTL_Locate is required.

C. BIOS 232 bit Entry Point Specification

In order for IOCTL_Locate to find the entry point for the BIOS, the BIOS 232-bit Service Directory is used. A description of the BIOS 232-bit Service Directory is described in Appendix C. The signature that Access Driver 246 will use when locating and executing BIOS functions shall be "_32_".

If the WinntEntry (BIOS232 Service Directory) structure is not found subject to the conditions stated above, the Access Driver 246 driver will fail at load time, and DriverEntry will indicate that it was unable to initialize as per Table 2.

D. Real-Time Clock Hardware Access

In order to implement the IOCTL_RTC_Read function, it is necessary to define the RTC's registers and methods of access. The RTC registers are located in the CMOS RAM's I/O address space. Only the RTC registers are shown in Table 3. The registers are accessed by outputting a CMOS physical memory address to port 0x70, and then reading the subject 8 bit register at port 0x71. The CMOS physical memory address is set to point to 0x0D after all RTC register have been read.

Table 3. RTC Registers

Register Name	CMOS Address, Comments
Year	9 - years since 1980
Month	8 - 1 = January
Date	7
Day of Week	6 - zero equals Sunday
Hour	4
Minute	2
Seconds	0

Figure 6A is a flowchart illustrating one embodiment of the initialization process of the present invention. Beginning from a start state, process 600 proceeds to block 610, the variables for the calling program (i.e., the I/O Manager 242) are initialized. Details of this initialization process 6100 are provided in Figure 6B and the accompanying text. Process 600 then proceeds to block 620 where it loads the access driver 246. Initialization of the access driver variables then occurs. During

the initialization process, two particularly essential initializations occur: (a) the BIOS shadow area 260 (which includes a BIOS Service Directory 62) located in physical memory 250 and (b) the BIOS data area 264, also located in physical memory 250, are both mapped into the access driver's 246 virtual memory (shown in figure 4 as the BIOS shadow area 270 (which includes the BIOS Service Directory 272) and the BIOS data area 274) of the access driver 246.

Process 600 then advances to block 630, where pointer initialization occurs. Details of block 630 are provided in Figure 6C and the accompanying text. The process 600 then advances to block 640, where initialization ends. Process 600 then terminates.

Figure 6B is a flowchart illustrating the details of block 610. Beginning from a start state, process 610 proceeds to block 612, where the calling program from the I/O Manager 242 allocates memory for a specified memory structure in the system buffer. The process 610 then advances to process block 614, where the calling program from the I/O Manager 242 determines the location of a number of BIOS functions, their corresponding entry points, lengths and offsets. In one embodiment, this is accomplished by entering the address field of the specified memory structure for the corresponding BIOS function with the virtual address of the BIOS function and by providing a 4-byte ASCII string identifying each BIOS function. Initialization of the calling program then terminates.

Figure 6C is a flowchart illustrating the details of process block 630 of Figure 6A. Beginning from a start state, the calling application program 232 or service 234, through the class driver, makes a call to IOCTL_Locate, as shown in block 632. In response, the access driver 246 performs a search for the BIOS Service Directory 272 header, as shown in process block 634. . Upon finding and validating the BIOS

Service Directory 272, the access driver 246 obtains the virtual address of the BIOS Service Directory 272 header, which provides the offset of the BIOS Service Directory 272 header virtual address from the base virtual address of the BIOS shadow area 270. The process 630 then returns control to the calling application program 232 or service 234.

Figure 7A is a flowchart illustrating the calling execution process of the present invention. Beginning from a start state, the process 700 proceeds to process block 710, where the calling program calls a BIOS function by providing to the access driver 246, the address of the BIOS function it wants to begin execution at. The process 700 then proceeds to process block 720, where the access driver 246 receives a dispatch call to a BIOS function via an IOCTL command from the I/O Manager 242 (see Figure 4). The process 700 then proceeds to process block 730, where the access driver 246 conducts a range check of the entry point address. In particular, the access driver 246 determines if the entry point address is within the range of addresses mapped to the BIOS shadow area, exclusive of the service directory header. If not, the access driver 246 indicates that the starting virtual address is not within the range of addresses mapped from the physical memory to the virtual memory. This may be indicated through the use of a flag. If the range check is successful, the process 700 proceeds to process block 740, where the access driver 246 executes the BIOS function called. The process 700 then terminates.

Figure 7B is a flowchart illustrating the details of process block 740 of Figure 7A. Beginning from a start state, the process 740 proceeds to process block 742, where the access driver 246 creates a register stack in a system buffer previously specified by the program from the I/O Manager 242. The process 730 then advances to process block 744, where the access driver 246 provides a pointer to the register stack which holds the address of the BIOS function to be executed. The process 740 then proceeds

to process block 746, where the calling program from the I/O Manager 242 calls and executes the function whose beginning address is indicated by the pointer, using its physical address as mapped in virtual memory. The process 740 then terminates.

An example of the utilization of the IOCTL_BIOS_EXEC function in the access driver 246 will now be provided. Initially, the application program 232 or service 234 makes a call to the access driver 246 using the command IOCTL_Locate. The data returned by the access driver 246 includes the BIOS Shadow Area Base Virtual Address, the BIOS Service Directory offset from the BIOS Shadow Area Base Virtual Address, and the BIOS Data Area Base Virtual Address.

The following act is then utilized to determine the existence of a BIOS service, its entry point, length and address offset. A calling program from the I/O Manager 242 first allocates memory for a register structure, such as IOC_EXEC1 and then fills in the biosFunction field of the structure with the virtual address given by the IOCTL_Locate function. The other register values are filled in as follows: a 4-byte ASCII string identifying the BIOS service is loaded into the eax register and a zero is loaded into the ebx register.

Next, the caller invokes the IOCTL_BIOS_Exec function of the access driver 246 with the contents of the IOC_EXEC1 structure copied into the system buffer for the IOCTL call. The BIOS function is then executed. The IOCTL_BIOS_Exec function of the access driver 246 returns, with register values for eax, ebx, ecx and edx each containing responses from the service directory. The calling program of the I/O Manager 242 then takes the information returned from the service directory and creates a biosFunction entry point and a structure in the system buffer. It then calls the BIOS function using the IOCTL_BIOS_Exec function in the access driver 246. Returned data are passed in the same IOC_EXEC1 structure.

Examples of the processes shown in Figures 6A, 6B, 7A and 7B are illustrated in Appendices D1-D3. In particular, Appendix D1 illustrates exemplary source code for the application program 232, service 234 or class driver 240, used in calling a BIOS function through the class driver 246. Appendices D2 and D3 illustrate exemplary source code for the access driver 246. Appendix D2 illustrates exemplary source code for executing a BIOS function in the shadow area, while Appendix D3 illustrates exemplary source code for creating a register stack and for calling the entry point for executing the BIOS function.

Through the use of the present invention, a system and method for accessing and executing the contents of physical memory from a virtual memory subsystem is provided. The system and method facilitates increased addressing capability for memory and input/output operations, and also allows execution of processor instructions in physical memory space.

Secure Utilization of BIOS services

Another aspect of the present invention includes a system and method for securely utilizing Basic Input and Output System (BIOS) services . In the following detailed description, the following terms are used to described the current invention:

- A "key" is an encoding and/or decoding parameter in accordance with conventional cryptographic algorithms such as Rivest, Shamir and Adleman (RSA), Data Encryption Algorithm (DEA) as specified in Data Encryption Standard (DES), and the like.

- A "key pair" includes a "private" key and a "public" key. A "private key" is held by the owner of the key pair and is used to generate digital signatures. A "public" key is widely published and is used to verify digital signatures. A public key is usually published in the form of a digital "certificate".

- A "digital signature" is a digital quantity that is generated using a digital message and a private key. A digital signature cannot be computed without knowing the private key. A digital signature can be verified using the digital message and a public key corresponding to the private key. Successful verification confirms that the digital message is indeed the one which was signed and that the signature was generated using the private key corresponding to the public key.

- A "certificate" is a digital message containing at least a public key, a private key, and a digital signature created using the private key.

Figure 8 is an overall functional block diagram illustrating the architecture of a processing system 1500 utilizing the system and method of the present invention. The processing system 1500 comprises an operating system 1505 which supports application programs 1510 and services 1515, Basic Input/Output System (BIOS) 1520 and system hardware 1525. BIOS 1520 is a collection of drivers, or software interfaces for hardware devices such as the console (keyboard and display), a generic printer, the auxiliary device (serial port), the computer's clock and the boot disk device. BIOS 1520 is typically embedded in non-volatile memory.

The operating system 1505 includes a class driver 1530 which interfaces with application programs 1510 and services 1515, and an I/O Manager 1535. The I/O Manager 1535 converts I/O requests from application programs 1510 and services 1515 (made via class driver 1530) into properly sequenced calls to various driver routines located in the kernel 1540. In particular, when the I/O Manager 1535 receives an I/O request, it uses the function codes of the request to call one of several dispatch routines in a driver located in the kernel 1540. The kernel 1540 provides hardware-independent functions, called system functions, that are accessed by means of a software interrupt. The functions provided by the kernel 1540 typically include file and directory management, memory management, character device input/output and time and date support, among others. In one embodiment, the operating system 1505 a Windows operating system. In alternate embodiments, the operating system 1505 includes the Solaris or the AIX operating systems or other operating systems based on demand-paged virtual memory subsystems.

The present invention provides an access driver 1545, located within the kernel 1540, which is responsible for interfacing with the ROM Application Programming Interface (RAPI) 1550 to access or update data located in BIOS 1520 or access system hardware data via the BIOS. RAPI 1550 generally provides an interface for securely utilizing BIOS services or functions. A more detailed description of RAPI is provided below.

In one preferred embodiment, the access driver 1545 comprises source code written in the C language. It is understood that other assembly languages may be

utilized in implementing the functions of access driver 1545. In one preferred embodiment, a set of entry-points or functions calls are available to application programs 1510, services 1515 or class driver 1530 which utilize access driver 1545. The access driver 1545 can be opened, closed, and can receive input/output ("I/O") control codes ("IOCTLs") through these entry points.

Figure 9 shows an illustrative sequence of interactive sequence between access driver 1545 and RAPI 1550 in accordance with one embodiment of the current invention. In a present system, a work session must be established between access driver 1545 and RAPI 1550 before access driver 1545 can send one or more service requests to RAPI 1550 to utilize BIOS services. To establish a work session with RAPI 550, access driver 1545 generates a session request (block 1605) and send the request to RAPI 1550 (block 1610).

The format of one embodiment of a session request 900 is shown in Figure 12. Each session request 900 includes a session operation code 905, a list of parameters 910, and a session signature 915. Session operation code 905 is a numerical value representing one type of session operation. Illustrative examples of session operations in one embodiment may include an operation to begin or establish a session and an operation to end or terminate the session. Each type of session operation may require a list of one or more parameters 910. In one embodiment, the list of parameters 910 may be a pointer to a memory location where the parameters reside. Each session request 900 also includes a session request signature

915 to prevent a foreign code segment, such as a computer virus, to capture and replay the request and corrupt the BIOS.

Figure 10 outlines the generation of a session request. In blocks 1705 and 1710, the session operation code representing a desired session operation and the list of parameters required for that desired session operation is inserted in the session request. Blocks 1715, 1720, and 1725 show the creation of a session request signature. In the art of cryptography, the act of creation of a digital signature for a message is known as "signing" the message. It should be noted that algorithms to sign a message or to create digital signatures for a message are known in the art. It should be further noted that existing algorithms for creating digital signatures generally include computing a hash value of the message to be signed and encrypting the hash value using a private key, as shown in blocks 1715, 1720, and 1725.

It is contemplated that the Digital Signature Algorithm ("DSA") proposed by the National Institute of Standards and Technology may be used. It is also contemplated that the Rivest, Shamir, and Adleman ("RSA") algorithm may be used. It should be noted, however, that other algorithms for generating digital signatures may also be employed in the present invention.

As shown in block 1715 of Figure 10, a session message is formed such that it contains the session operation code and the list of parameters. In block 1720, a hash value for the session message is computed. It should be noted that algorithms to compute hash values are well known in the art. Those skilled in the art will recognize that a hash function suitable for use in embodiments of the present

invention is one that can compute one-way and collision-free hash values. In block 1725, a session request signature is generated by encrypting the computed hash value for the session message using a private key stored in a current authority certificate.

An authority certificate generally contains sufficient information to enable system components, such as access driver 1545 and RAPI 1550 (shown in Figure 9), to generate secured session or service requests and to verify the integrity of those requests. Figure 11 shows the format of an authority certificate 800 in accordance with one embodiment of the current invention. The exemplary authority certificate includes at least the following fields: a public key 805, a private key 810, and a certificate signature 815. As described later, information stored in the authority certificate is used to enforce security limits of session and service requests.

Returning to block 1615 in Figure 9, RAPI 1550 establishes a session as specified in the session request sent by access driver 1545 in block 1610. The acts required to establish a session are outlined in Figure 14. In block 1105, a message is constructed from the session operation code and list of parameters available in a session request received from the access driver. A hash value for the constructed message is computed (block 1110). The session request signature is extracted from the session request and decrypted using the public key included in RAPI's copy of the current authority certificate (block 1115). As shown later, RAPI is responsible for generating and supplying authority certificates to the access driver. RAPI, however, also maintains a copy of the most current authority certificates for its own use. In block 1120, the decrypted session signature is compared to the computed hash value

for the constructed message. If the computed hash value equals the decrypted session request signature, RAPI proceeds to start a session (block 1125).

Returning to Figure 9, after it establishes a session, RAPI 1550 generates a new authority certificate (block 1620). As stated above, RAPI 1550 maintains a copy of the new authority certificate for its own use. RAPI 1550 replaces the existing authority certificate with the new certificate. After the replacement, the new certificate becomes the most current certificate.

Figure 18 shows the process of generating an authority certificate. To generate an authority certificate, a new key pair is obtained from a provided cryptographic engine (block 2505). As defined above, a new key pair includes a public key and private key. The new keys are inserted in the new authority certificate (blocks 2510 and 2515). A certificate message is formed such that it includes both the new public and private keys (block 2520). A hash value for the certificate message is computed (block 2525). A certificate signature is generated using the hash value for the certificate message and the new private key (block 2530). The certificate signature is then inserted into the new authority certificate (block (2535).

Returning to Figure 9, RAPI 1550 sends the new authority certificate back to the access driver 1545 (block 1625). Upon receiving the new authority certificate, access driver 1545 updates the current authority certificate with information in the new certificate (block 1630). Accordingly, the information in the new authority certificate will be used to generate subsequent service requests. In block 1635, access driver 1545 generates a service request to invoke a function provided by RAPI 1550.

Figure 13 shows the format of a service request 1000 in accordance with one embodiment of the current invention. Each service request 1000 includes a service operation code 1005, a list of parameters 1010, and a service request signature 1015. Service operation code 1005 is a numerical value representing one type of service operation. Illustrative examples of service operations in one embodiment may include operations to read or write data stored in non-volatile memory. Each type of service operation may require one or more parameters that are included in the list of parameters. In one embodiment, the list of parameters may be a pointer to a memory location where a bundle of parameters reside. Each service request 1000 further includes a service request signature 1015 to prevent a foreign code segment, such as a computer virus, to capture and replay the service request to disable or cause havoc to the system.

Figure 15 outlines the generation of a service request in accordance with one embodiment of the current invention. In blocks 1205 and 1210, the service operation code representing a desired service operation to be performed and the list of parameters required for that desired operation are inserted in the service request. Blocks 1215, 1220, and 1225 show the creation of a service request signature. Once created, the service request signature is inserted in the service request (block 1230).

Returning to Figure 9, access driver 1545 sends the service request generated in block 1635 to RAPI 1550 (block 1640). Upon receiving the service request, RAPI 1550 processes the request (block 1645). Figure 16 shows the acts required in processing of a service request in accordance with one embodiment of the current

invention. In block 1305, a message is constructed from the service operation code and list of parameters available in the service request received from access driver. A hash value for the constructed message is computed (block 1310). The service request signature is extracted from the session request and decrypted using the public included in RAPI's copy of the current authority certificate (block 1315). In block 1320, the decrypted session signature is compared to the computed hash value for the constructed message. If the computed hash value equals the decrypted session request signature, RAPI performs the service specified in the service request (block 1325). Otherwise, the specified service will not be performed.

Returning to Figure 9, after it processes a service request, RAPI 1550 generates a new authority certificate (block 1650). As described above, Figure 15 shows the process of generating an authority certificate. RAPI 1550 maintains a copy of the new authority certificate for its own use. RAPI 1550 also sends a copy of the new authority certificate back to access driver 1545.

Upon receiving the new authority certificate, access driver 1545 updates the current authority certificate with information in the new certificate (block 1630). Accordingly, the information in the new authority certificate will be used to generate subsequent service requests. In block 1635, access driver 1545 generates a session request to request RAPI 1550 to end the current session. As described above, Figure 10 outlines the acts involved in generating a session request. Following the generation of a session request, access driver 1545 sends the request to RAPI 1550.

Upon receipt of the session request to end or terminate the session, RAPI 1550 ends the session (block 1680). Figure 17 shows the acts involved in ending the current session. In block 1405, a message is constructed from the session operation code and list of parameters available in a session request received from the access driver. A hash value for the constructed message is computed (block 1410). The session request signature is extracted from the session request and decrypted using the public key included in RAPI's current authority certificate (block 1415). In block 1420, the decrypted session signature is compared to the computed hash value for the constructed message. If the computed hash value equals the decrypted session request signature, RAPI proceeds to end the current session (block 1425).

Returning to Figure 9, after it ends a session, RAPI 1550 generates a new authority certificate (block 1685). As described above, Figure 15 shows the process of generating an authority certificate. RAPI 1550 then sends the new authority certificate back to the access driver 1545 (block 1690). Upon receiving the new authority certificate, access driver 1545 updates the current authority certificate with information in the new certificate (block 1695). Accordingly, the information in the new authority certificate will be used to generate requests in subsequent sessions.

Figure 9 shows that access driver generates only one service request in a work session. In practice, a plurality of service requests may be generated and sent to RAPI 1550 between each work session.

In summary, the present invention requires the inclusion of a digital signature in session and service requests as a security measure to prevent

components foreign to the system, such as viruses, to invoke BIOS functions or services. Furthermore, each successive session or service request includes a digital signature that is generated using a new private key to prevent foreign components from capturing and replaying the session and/or service requests and causing adverse effects to the system. Thus, the security measure employed in the present invention ensures safe and secured utilization of BIOS functions.

Although the present invention has been described in terms of certain preferred embodiments, other embodiments apparent to those of ordinary skill in the art are also within the scope of this invention. Accordingly, the scope of the invention is intended to be defined only by the claims which follow.

CLAIMS:

What is claimed is:

1. A system for accessing and executing instruction sequences in a physical memory from a virtual memory in a processor-based system, comprising:

a memory for storing instruction sequences by which the processor-based system is processed, the memory including a physical memory and a virtual memory; and

a processor for executing the stored instruction sequences; and

wherein the stored instruction sequences include process steps to cause the processor to: (a) map a plurality of predetermined instruction sequences from the physical memory to the virtual memory; (b) determine an offset to one of said plurality of predetermined instruction sequences in the virtual memory; (c) receive an instruction to execute the one of said plurality of predetermined instruction sequences; (d) transfer control to the one of said plurality of predetermined instruction sequences; and (e) process the one of said plurality of predetermined instruction sequences from the virtual memory.

2. The system of Claim 1, wherein in step (c), the instruction is made from an application program.

3. The system of Claim 1, wherein in step (c), the instruction is made from a class driver.

4. The system of Claim 1, wherein step (a) comprises the steps of:

(a.1) mapping a plurality of BIOS instruction sequences from the physical memory to the virtual memory, said BIOS instruction sequences including a BIOS service directory; and

(a.2) mapping BIOS data from the physical memory to the virtual memory.

5. The system of Claim 4, wherein step (b) comprises the steps of:

(b.1) determining a starting virtual address of the BIOS service directory; and

(b.2) determining a starting virtual address of one of the plurality of BIOS instruction sequences by reference to the BIOS service directory.

6. The system of Claim 5, wherein step (d) comprises the steps of:

(d.1) creating a register stack in a memory location;

(d.2) identifying a location of the starting virtual address of one of the plurality of BIOS instruction sequences in the register stack; and

(d.3) transferring control to the one of the plurality of BIOS instruction sequences.

7. The system of Claim 6, wherein in step (d.1), the memory location is a buffer located in a dynamic random access memory (DRAM).

8. The system of Claim 6, wherein in step (d.1), the memory location is a buffer located in a main memory.

9. The system of Claim 6, wherein step (e) comprises the steps of:

(e.1) determining if the starting virtual address is within a range of addresses mapped from the physical memory to the virtual memory; and

(e.2) if so, executing the one of the plurality of BIOS instruction sequences from the virtual memory, otherwise indicating that the starting virtual address is not within the range of addresses mapped from the physical memory to the virtual memory.

10. A method for accessing and executing instruction sequences in physical memory from virtual memory in a processor-based system, comprising the steps of:

(a) mapping a plurality of predetermined instruction sequences from the physical memory to the virtual memory;

(b) determining an offset to one of said plurality of predetermined instruction sequences in the virtual memory;

(c) receiving an instruction to execute the one of said plurality of predetermined instruction sequences;

(d) transferring control to the one of said plurality of predetermined instruction sequences; and

(e) processing the one of said plurality of predetermined instruction sequences from the virtual memory.

11. The method of Claim 10, wherein in step (c), the instruction is made from an application program.

12. The method of Claim 10, wherein in step (c), the instruction is made from a class driver.

13. The method of Claim 10, wherein step (a) comprises the steps of:

(a.1) mapping a plurality of BIOS instruction sequences from the physical memory to the virtual memory, said BIOS instruction sequences including a BIOS service directory; and

(a.2) mapping BIOS data from the physical memory to the virtual memory.

14. The method of Claim 13, wherein step (b) comprises the steps of:

(b.1) determining a starting virtual address of the BIOS service directory; and

(b.2) determining a starting virtual address of one of the plurality of BIOS instruction sequences by reference to the BIOS service directory.

15. The method of Claim 14, wherein step (d) comprises the steps of:

(d.1) creating a register stack in a memory location;

(d.2) identifying a location of the starting virtual address of one of the plurality of BIOS instruction sequences in the register stack; and

(d.3) transferring control to the one of the plurality of BIOS instruction sequences.

16. The method of Claim 15, wherein in step (d.1), the memory location is a buffer located in a dynamic random access memory (DRAM).

17. The method of Claim 15, wherein in step (d.1), the memory location is a buffer located in a main memory.

18. The method of Claim 15, wherein step (e) comprises the steps of:

(e.1) determining if the starting virtual address is within a range of addresses mapped from the physical memory to the virtual memory; and

(e.2) if so, executing the one of the plurality of BIOS instruction sequences from the virtual memory, otherwise indicating that the starting virtual address is not within the range of addresses mapped from the physical memory to the virtual memory.

19. Computer-executable process steps for accessing and executing instruction sequences in physical memory from virtual memory in a processor-based system, the process steps including:

(a) mapping a plurality of predetermined instruction sequences from the physical memory to the virtual memory;

(b) determining an offset to one of said plurality of predetermined instruction sequences in the virtual memory;

(c) receiving an instruction to execute the one of said plurality of predetermined instruction sequences;

(d) transferring control to the one of said plurality of predetermined instruction sequences; and

(e) processing the one of said plurality of predetermined instruction sequences from the virtual memory.

20. Computer-executable process steps of Claim 19, wherein in step (c), the instruction is made from an application program.

21. Computer-executable process steps of Claim 19, wherein step (a) comprises the steps of:

(a.1) mapping a plurality of BIOS instruction sequences from the physical memory to the virtual memory, said BIOS instruction sequences including a BIOS service directory; and

(a.2) mapping BIOS data from the physical memory to the virtual memory.

22. Computer-executable process steps of Claim 21, wherein step (b) comprises the steps of:

(b.1) determining a starting virtual address of the BIOS service directory; and

(b.2) determining a starting virtual address of one of the plurality of BIOS instruction sequences by reference to the BIOS service directory.

23. Computer-executable process steps of Claim 22, wherein step (d) comprises the steps of:

(d.1) creating a register stack in a memory location;

(d.2) identifying a location of the starting virtual address of one of the plurality of BIOS instruction sequences in the register stack; and

(d.3) transferring control to the one of the plurality of BIOS instruction sequences.

24. Computer-executable process steps of Claim 23, wherein in step (d.1), the memory location is a buffer located in a dynamic random access memory (DRAM).

25. Computer-executable process steps in Claim 23, wherein in step (d.1), the memory location is a buffer located in a main memory.

26. Computer-executable process steps of Claim 23, wherein step (e) comprises the steps of:

(e.1) determining if the starting virtual address is within a range of addresses mapped from the physical memory to the virtual memory; and

(e.2) if so, executing the one of the plurality of BIOS instruction sequences from the virtual memory, otherwise that the starting virtual address is not within the range of addresses mapped from the physical memory to the virtual memory.

27. A system for accessing instruction sequences in a physical memory from a virtual memory in a processor-based system, comprising:

a memory for storing instruction sequences by which the processor-based system is processed, the memory including a physical memory and a virtual memory; and

a processor for executing the stored instruction sequences; and

wherein the stored instruction sequences include process steps to cause the processor to: (a) map a plurality of predetermined instruction sequences from the physical memory to the virtual memory; (b) determine an offset to one of said plurality of predetermined instruction sequences in the virtual memory; (c) receive an instruction to execute the one of said plurality of predetermined instruction sequences; (d) transfer control to the one of said plurality of predetermined instruction sequences; and (e) process the one of said plurality of predetermined instruction sequences from the virtual memory.

28. The system of Claim 27, wherein step (a) comprises the steps of:

(a.1) mapping a plurality of BIOS instruction sequences from the physical memory to the virtual memory, said BIOS instruction sequences including a plurality of BIOS read only memory (ROM) instruction sequences and a BIOS service directory; and

(a.2) mapping BIOS data from the physical memory to the virtual memory.

29. The system of Claim 28, wherein step (b) comprises the steps of:

(b.1) determining a starting virtual address of the BIOS service directory; and

(b.2) determining a starting virtual address of one of the plurality of BIOS instruction sequences by reference to the BIOS service directory.

30. The system of Claim 29, wherein step (d) comprises the steps of:

(d.1) creating a register stack in a memory location and;

(d.2) identifying a location of the starting virtual address of one of the plurality of BIOS ROM instruction sequences in the register stack.

31. The system of Claim 30, wherein step (e) comprises the steps of:

(e.1) determining if the starting virtual address is within a range of addresses mapped from the physical memory to the virtual memory; and

(e.2) if so, reading the one of the plurality of BIOS ROM instruction sequences from the virtual memory, otherwise indicating that the starting virtual address is not within the range of addresses mapped from the physical memory to the virtual memory.

32. A method for accessing instruction sequences in physical memory from virtual memory in a processor-based system, comprising the steps of:

(a) mapping a plurality of predetermined instruction sequences from the physical memory to the virtual memory;

(b) determining an offset to one of said plurality of predetermined instruction sequences in the virtual memory;

(c) receiving an instruction to execute the one of said plurality of predetermined instruction sequences;

(d) transferring control to the one of said plurality of predetermined instruction sequences; and

(e) processing the one of said plurality of predetermined instruction sequences from the virtual memory.

33. The method of Claim 32, wherein step (a) comprises the steps of:

(a.1) mapping a plurality of BIOS instruction sequences from the physical memory to the virtual memory, said BIOS instruction sequences including a plurality of BIOS read only memory (ROM) instruction sequences and a BIOS service directory; and

(a.2) mapping BIOS data from the physical memory to the virtual memory.

34. The method of Claim 33, wherein step (b) comprises the steps of:

(b.1) determining a starting virtual address of the BIOS service directory; and

(b.2) determining a starting virtual address of one of the plurality of BIOS instruction sequences by reference to the BIOS service directory.

35. The method of Claim 34, wherein step (d) comprises the steps of:

(d.1) creating a register stack in a memory location; and

(d.2) identifying a location of the starting virtual address of one of the plurality of BIOS ROM instruction sequences in the register stack.

36. The method of Claim 35, wherein step (e) comprises the steps of:

(e.1) determining if the starting virtual address is within a range of addresses mapped from the physical memory to the virtual memory; and

(e.2) if so, reading the one of the plurality of BIOS ROM instruction sequences from the virtual memory, otherwise indicating that the starting virtual address is not within the range of addresses mapped from the physical memory to the virtual memory.

37. A system to securely utilize Basic Input and Output System (BIOS) services, comprising:

an access driver to generate a service request to utilize BIOS services, the service request including a service request signature created using a private key in a cryptographic key pair;.and

an interface to verify the service request signature using a public key in the cryptographic key pair to ensure the integrity of the service request.

38. The system of Claim 37, wherein:

the access driver generates a session request to establish a session with the interface; and

the session request includes a session request signature created using a private key in a cryptographic key pair.

39. The system of Claim 37, wherein:

the access driver generates a session request to end the session with the interface; and

the session request includes a session request signature created using a private key in a cryptographic key pair.

40. The system of Claim 37, wherein:

the interface generates an authority certificate and sends the authority certificate to the access driver after receiving a session request; and

the access driver uses information included in the authority certificate to generate subsequent session requests.

41. The system of Claim 40, wherein the authority certificate includes a new public key.

42. The system of Claim 40, wherein the authority certificate includes a new private key.

43. The system of Claim 40, wherein the authority certificate includes a certificate signature.

44. The system of Claim 37, wherein:

the interface generates an authority certificate and sends the authority certificate to the access driver after receiving the service request; and

the access driver uses information in the authority certificate to generate subsequent service requests.

45. A method to securely invoke Basic Input and Output System (BIOS) services, comprising:

creating a service request to invoke BIOS services;

signing the service request with a service request signature generated using a private key in a cryptographic key pair; and

verifying the service request signature using a public key in the cryptographic key pair to ensure the integrity of the service request.

46. The method of Claim 45, further comprising:

creating an authority certificate that includes a new private key and a new public key after processing the service request;

signing a subsequent service request with a service request signature generated using the new private key; and

verifying the service request signature of the subsequent service request using the new public key.

47. The method of Claim 45, further comprising:

performing a BIOS service indicated by a service operation code included in the service request.

48. The method of Claim 45, further comprising:

creating a session request to establish a session with a ROM Application Program Interface (RAPI);

signing the session request with a session request signature generated using a private key in a cryptographic key pair; and

verifying the session request signature using a public key in the cryptographic key pair to ensure the integrity of the session request.

49. The method of Claim 48, further comprising:

creating an authority certificate that includes a new private key and a new public key after processing the session request;

signing a subsequent session request with a session request signature generated using the new private key; and

verifying the session request signature of the subsequent session request using the new public key.

50. The method of Claim 45, further comprising:

creating a session request to end a session with a ROM Application Program Interface (RAPI);

signing the session request with a session request signature generated using a private key in a cryptographic key pair; and

verifying the session request signature using a public key in the cryptographic key pair to ensure the integrity of the session request.

51. A computer program embodied on a computer-readable medium to securely utilize Basic Input and Output System (BIOS) services, comprising:

an access driver to generate a service request to utilize BIOS services, the service request including a service request signature created using a private key in a cryptographic key pair; and

an interface to verify the service request signature using a public key in the cryptographic key pair to ensure the integrity of the service request.

52. A computer data signal embodied in a data stream, comprising:

an access driver to generate a service request to utilize BIOS services, the service request including a service request signature created using a private key in a cryptographic key pair; and

an interface to verify the service request signature using a public key in the cryptographic key pair to ensure the integrity of the service request.

ABSTRACT

In accordance with one aspect of the current invention, the system comprises a memory for storing instruction sequences by which the processor-based system is processed, where the memory includes a physical memory and a virtual memory. The system also comprises a processor for executing the stored instruction sequences. The stored instruction sequences include process acts to cause the processor to: map a plurality of predetermined instruction sequences from the physical memory to the virtual memory, determine an offset to one of the plurality of predetermined instruction sequences in the virtual memory, receive an instruction to execute the one of the plurality of predetermined instruction sequences, transfer control to the one of the plurality of predetermined instruction sequences, and process the one of the plurality of predetermined instruction sequences from the virtual memory. In accordance with another aspect of the present invention, the system includes an access driver to generate a service request to utilize BIOS services such that the service request contains a service request signature created using a private key in a cryptographic key pair. The system also includes an interface to verify the service request signature using a public key in the cryptographic key pair to ensure integrity of the service request.

FIG. 1

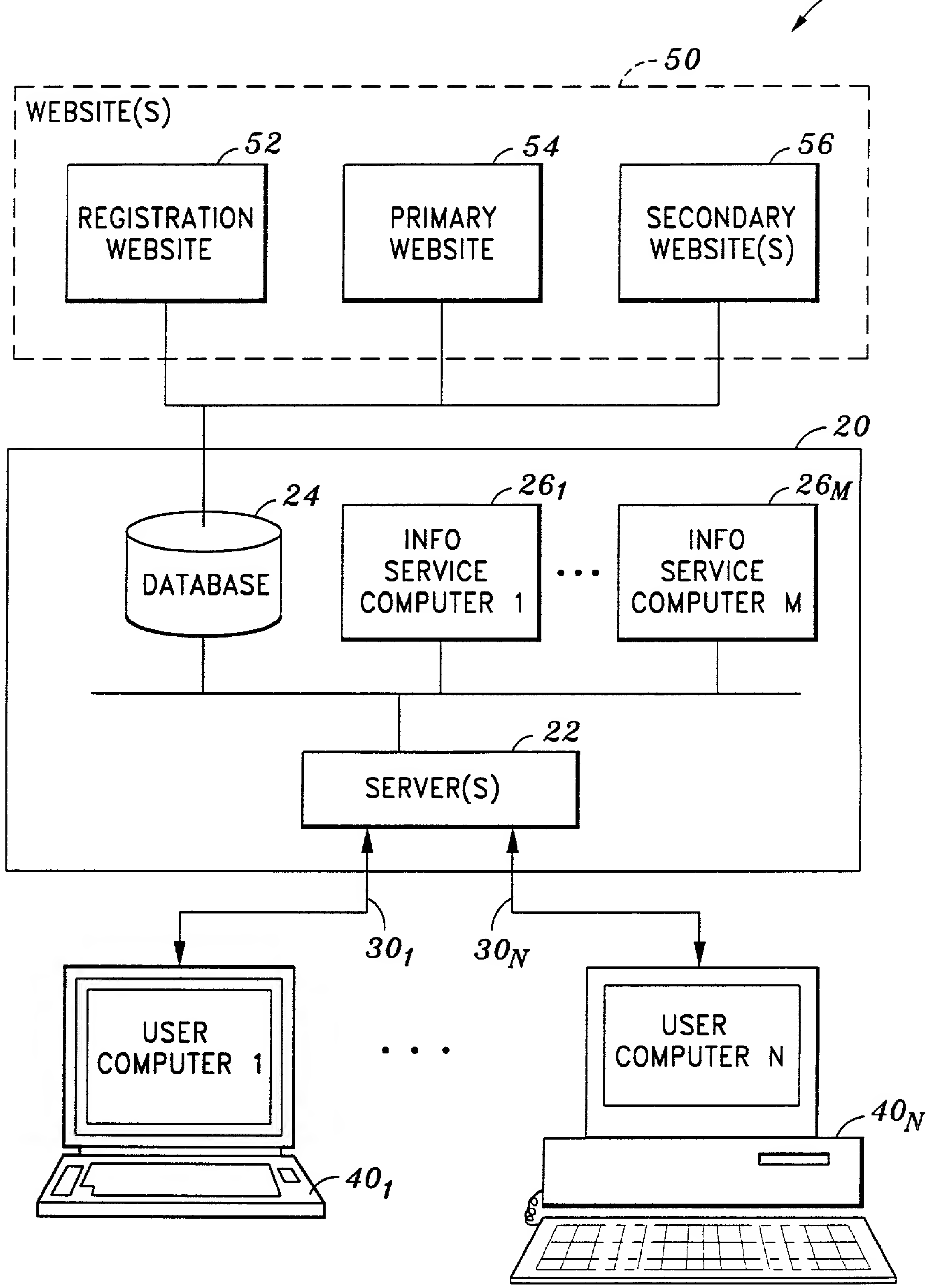
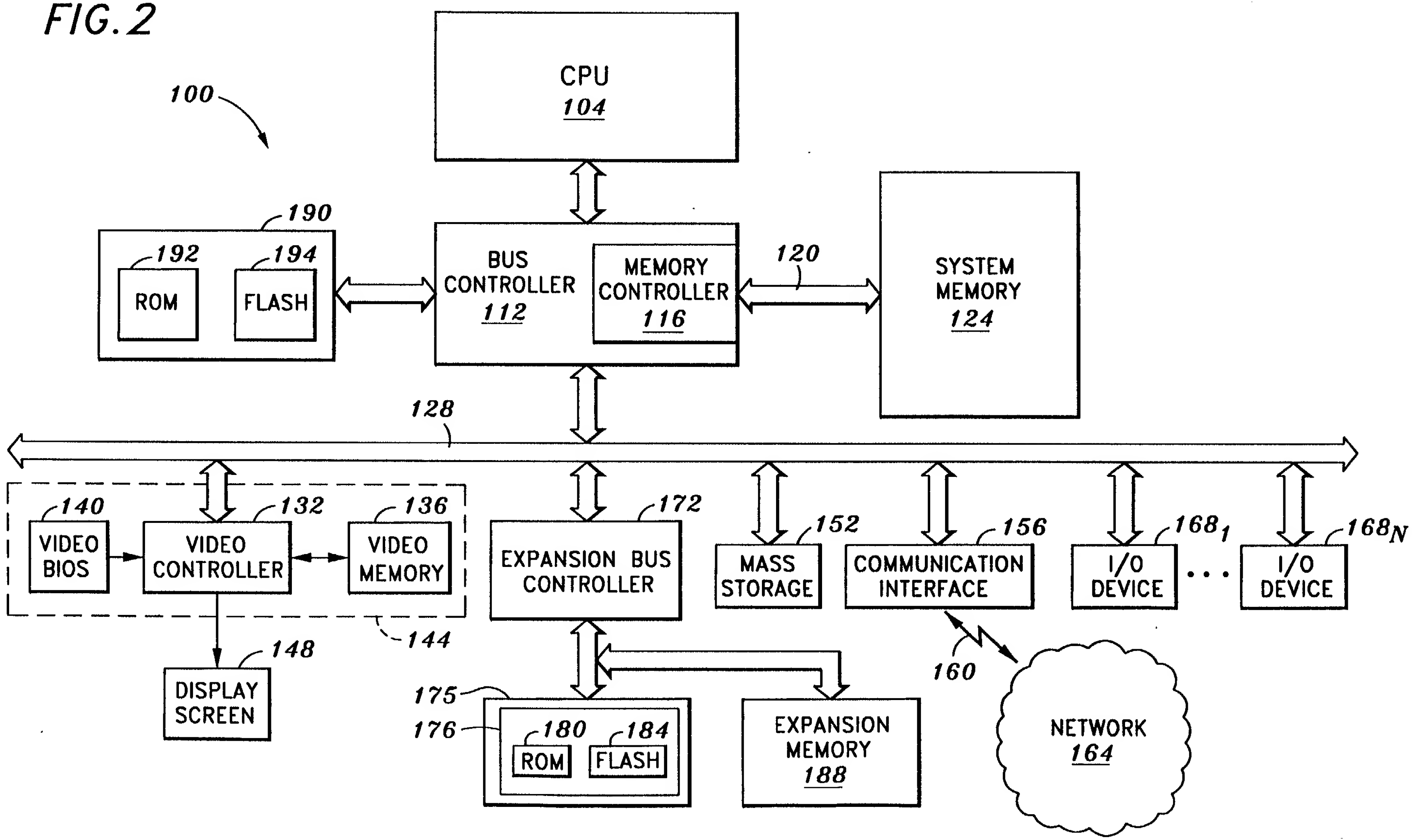

10
50
WEBSITE(S)
52
REGISTRATION WEBSITE
54
PRIMARY WEBSITE
56
SECONDARY WEBSITE(S)
20
24
DATABASE
26_1
INFO SERVICE COMPUTER 1
26_M
INFO SERVICE COMPUTER M
22
SERVER(S)
30_1
30_N
USER COMPUTER 1
USER COMPUTER N
40_1
40_N

FIG. 2



100
CPU
104
190
192
ROM
194
FLASH
BUS
CONTROLLER
112
MEMORY
CONTROLLER
116
120
SYSTEM
MEMORY
124
128
140
VIDEO
BIOS
132
VIDEO
CONTROLLER
136
VIDEO
MEMORY
144
148
DISPLAY
SCREEN
172
EXPANSION BUS
CONTROLLER
175
176
180
ROM
184
FLASH
EXPANSION
MEMORY
188
152
MASS
STORAGE
156
COMMUNICATION
INTERFACE
160
NETWORK
164
168₁
I/O
DEVICE
168N
I/O
DEVICE

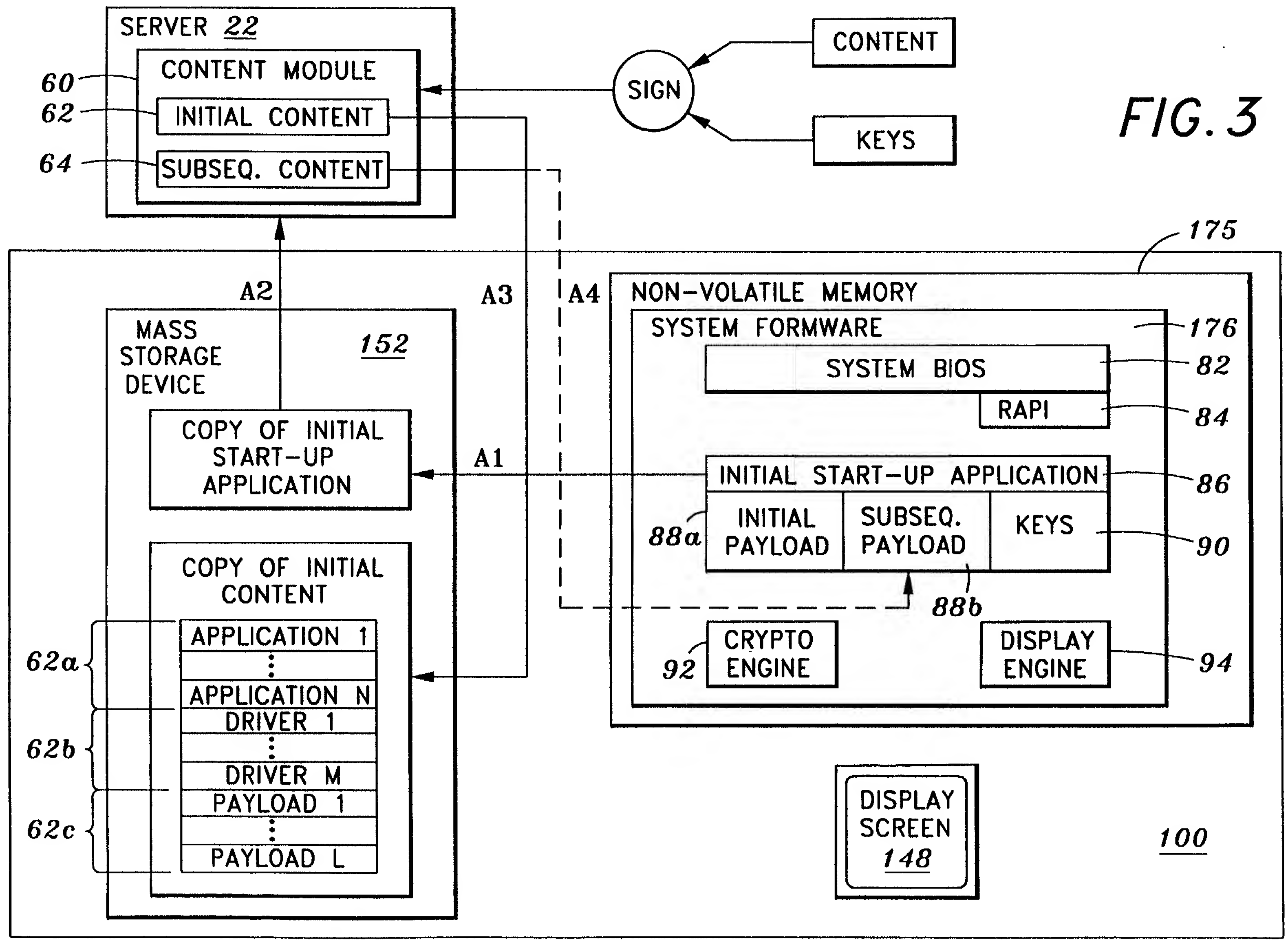

SERVER 22
CONTENT MODULE
INITIAL CONTENT
SUBSEQ. CONTENT
60
62
64
SIGN
CONTENT
KEYS
A2
A3
A4
A1
MASS STORAGE DEVICE
152
COPY OF INITIAL START-UP APPLICATION
COPY OF INITIAL CONTENT
APPLICATION 1
APPLICATION N
DRIVER 1
DRIVER M
PAYLOAD 1
PAYLOAD L
62a
62b
62c
175
NON-VOLATILE MEMORY
SYSTEM FORMWARE
176
SYSTEM BIOS
82
RAPI
84
INITIAL START-UP APPLICATION
86
INITIAL PAYLOAD
SUBSEQ. PAYLOAD
KEYS
88a
88b
90
CRYPTO ENGINE
92
DISPLAY ENGINE
94
DISPLAY SCREEN 148
100

FIG. 3

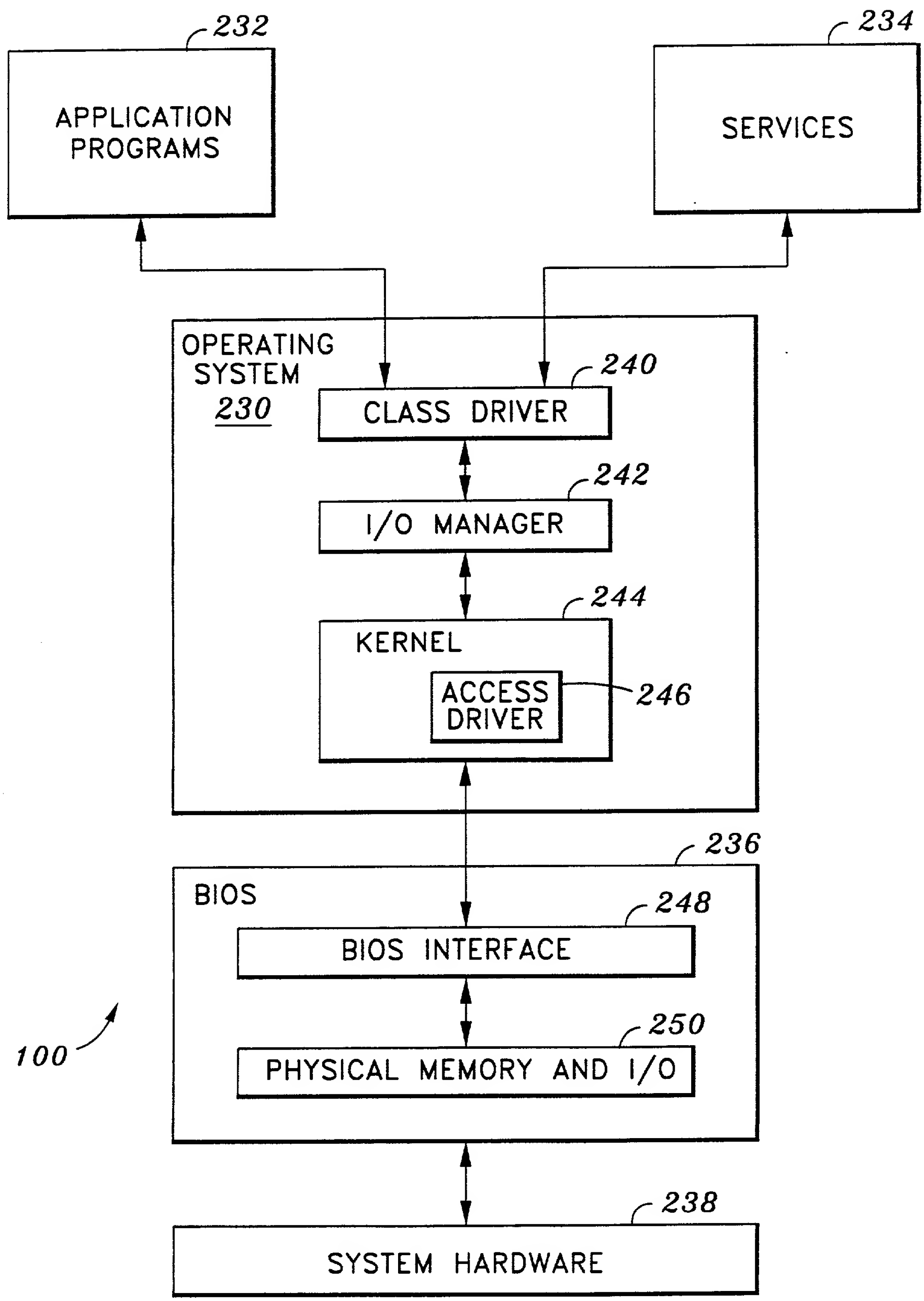

232
APPLICATION PROGRAMS
234
SERVICES
OPERATING SYSTEM
230
CLASS DRIVER
240
I/O MANAGER
242
KERNEL
244
ACCESS DRIVER
246
236
BIOS
BIOS INTERFACE
248
PHYSICAL MEMORY AND I/O
250
100
SYSTEM HARDWARE
238

FIG. 4

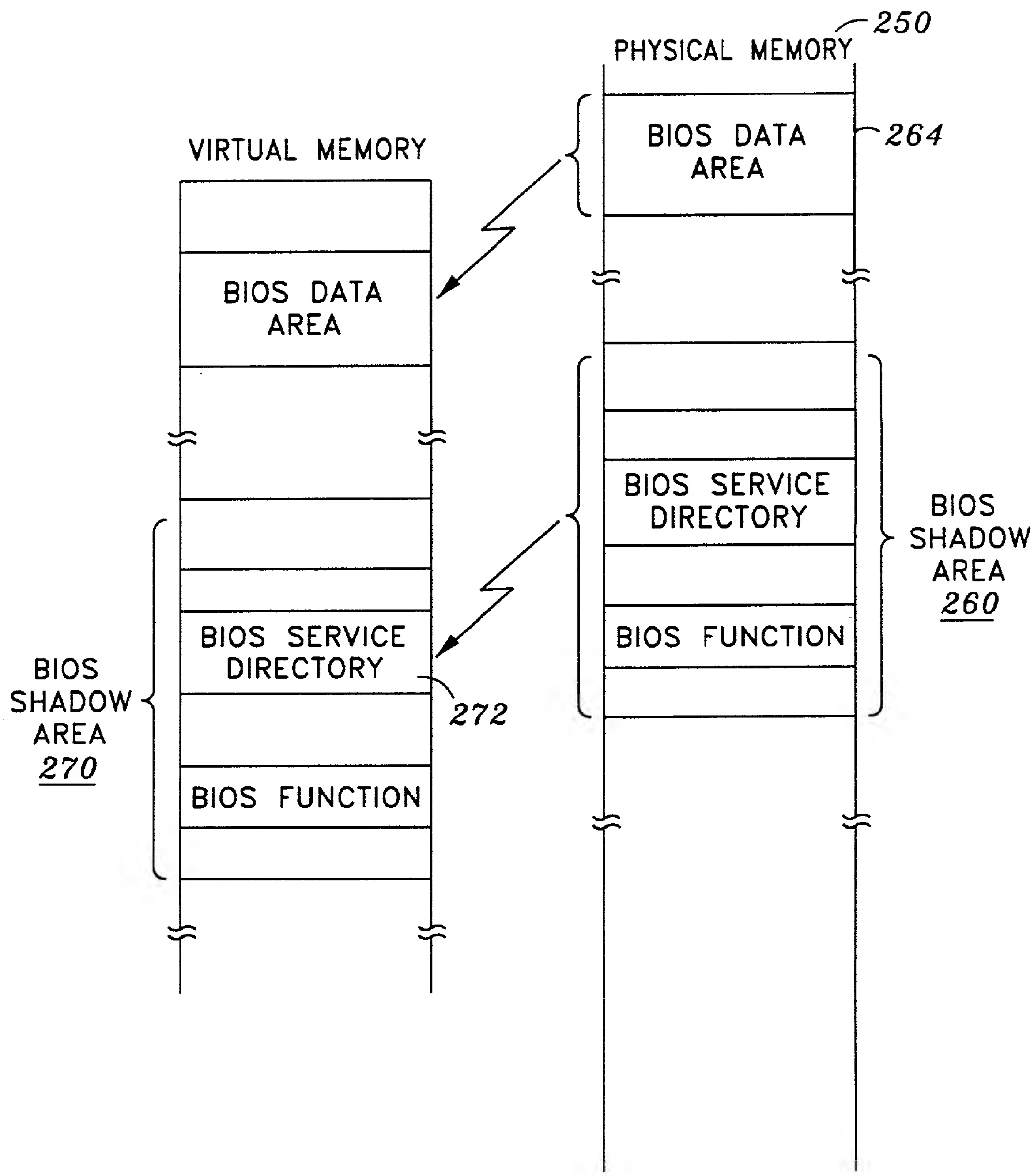

250
PHYSICAL MEMORY
264
BIOS DATA AREA
VIRTUAL MEMORY
BIOS DATA AREA
BIOS SERVICE DIRECTORY
BIOS FUNCTION
BIOS SHADOW AREA
260
BIOS SERVICE DIRECTORY
272
BIOS FUNCTION
BIOS SHADOW AREA
270

FIG. 5

INITIALIZATION PROCESS

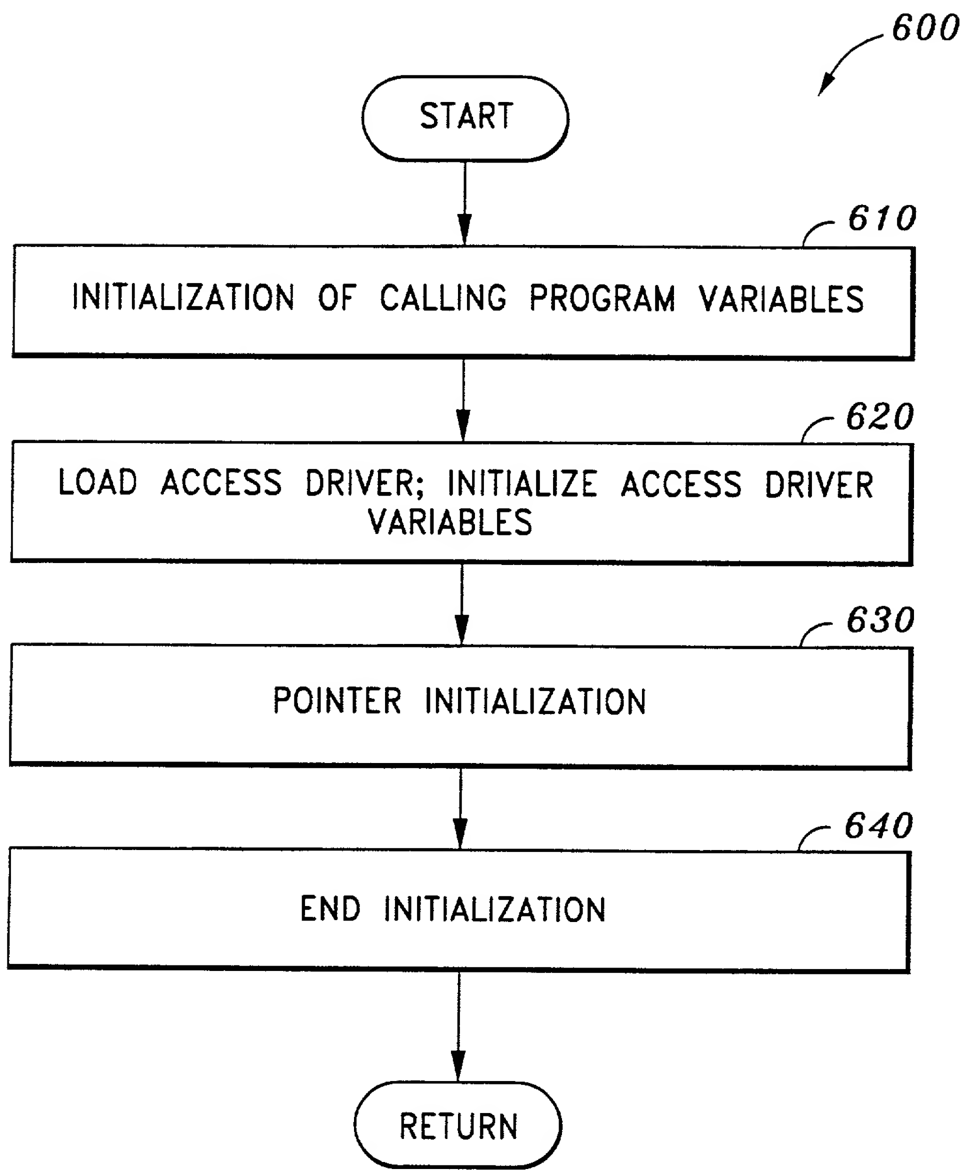


FIG. 6A

INITIALIZATION OF CALLING PROGRAM

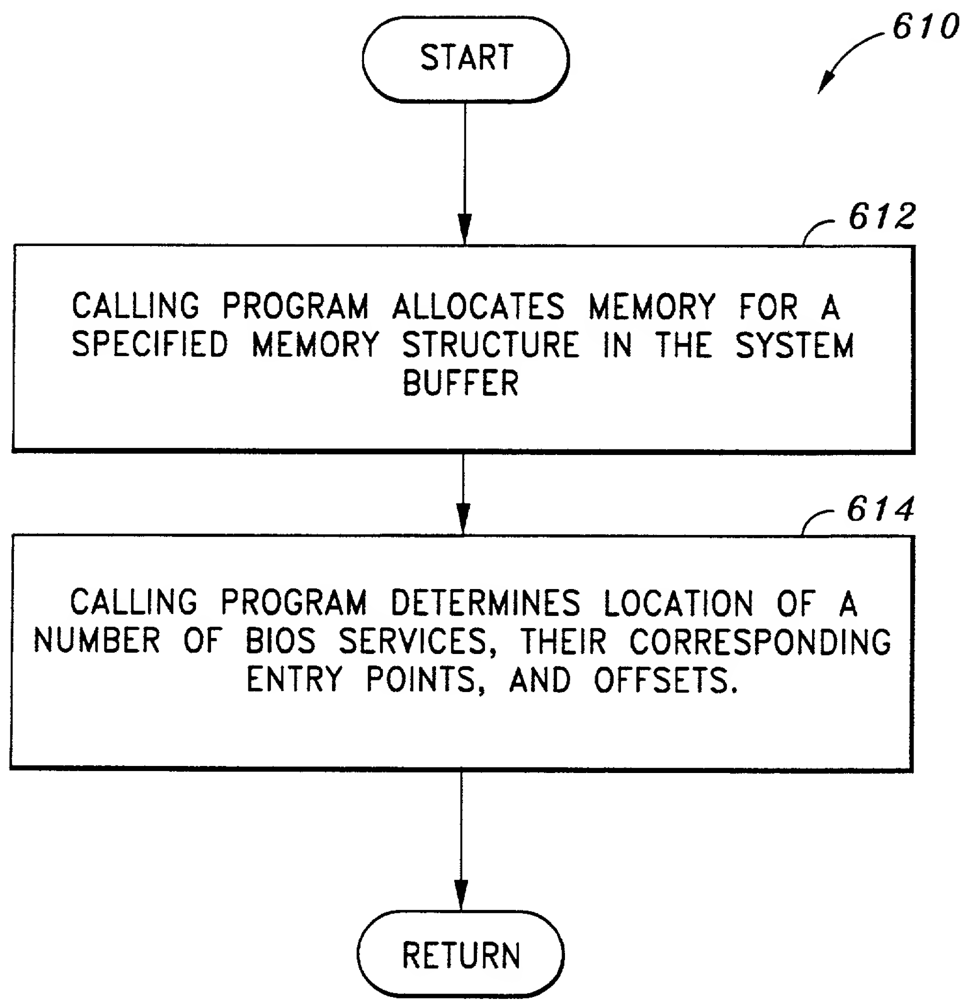


FIG. 6B

POINTER INITIALIZATION PROCESS

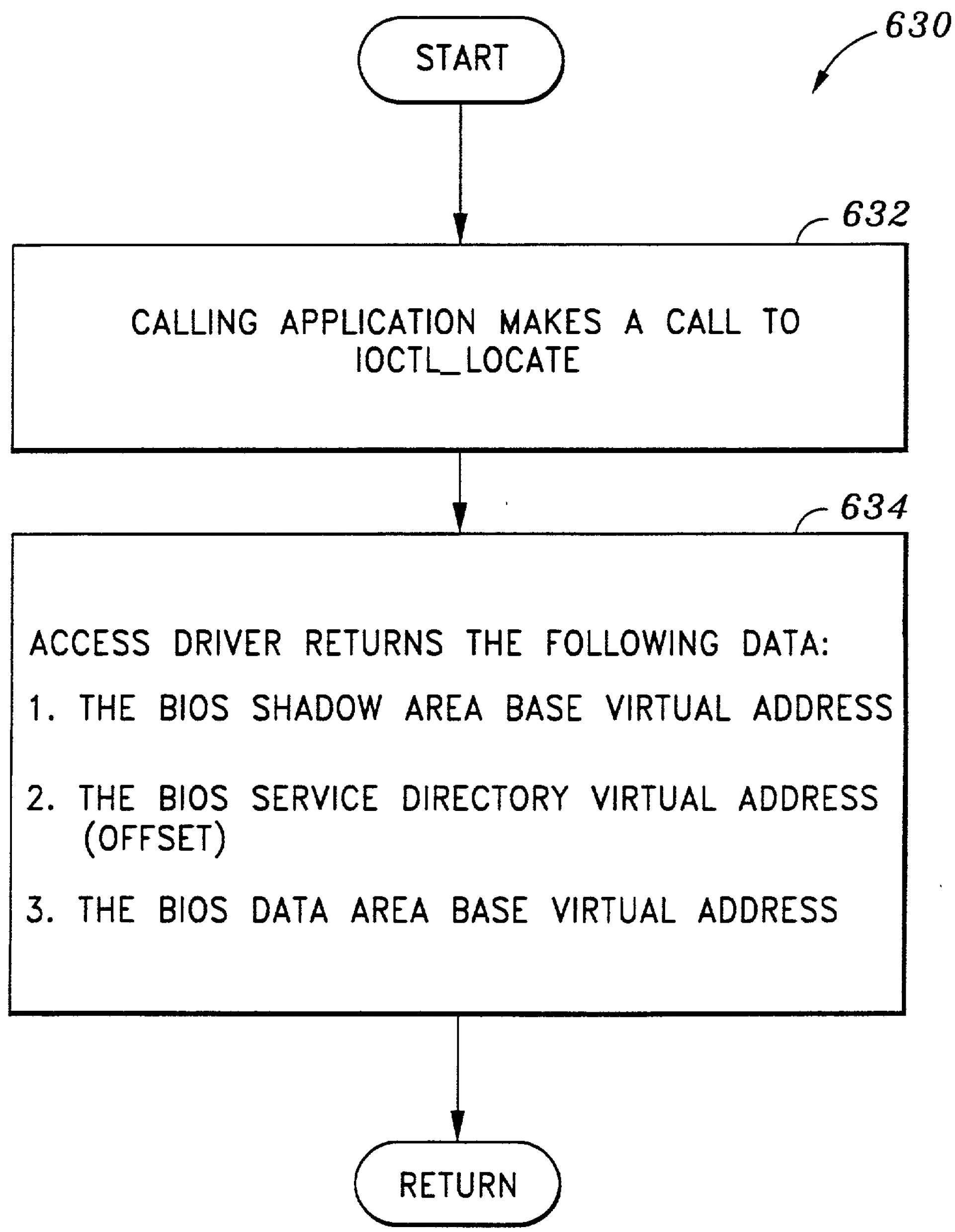


FIG. 6C

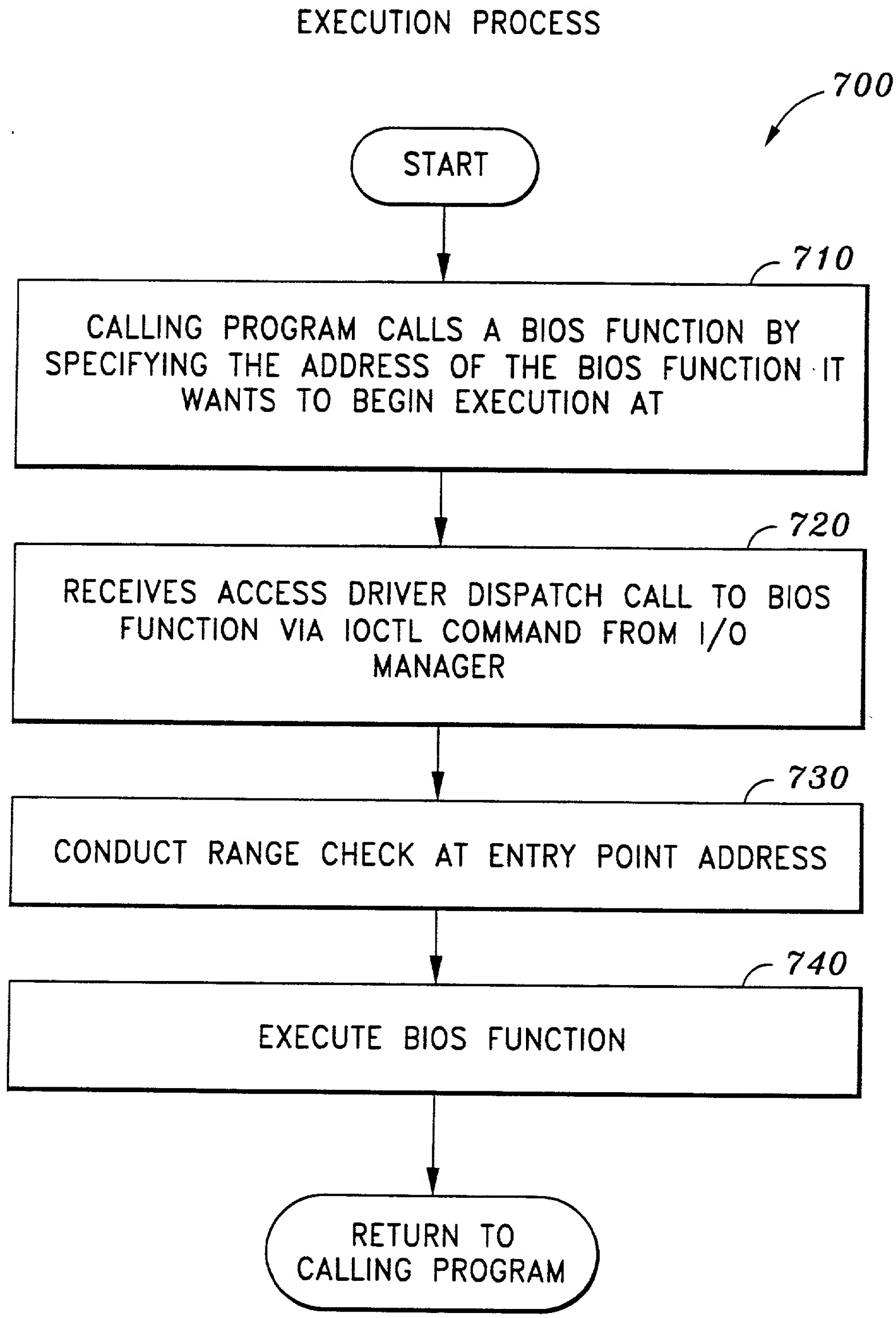

EXECUTION PROCESS
700
START
710
CALLING PROGRAM CALLS A BIOS FUNCTION BY SPECIFYING THE ADDRESS OF THE BIOS FUNCTION IT WANTS TO BEGIN EXECUTION AT
720
RECEIVES ACCESS DRIVER DISPATCH CALL TO BIOS FUNCTION VIA IOCTL COMMAND FROM I/O MANAGER
730
CONDUCT RANGE CHECK AT ENTRY POINT ADDRESS
740
EXECUTE BIOS FUNCTION
RETURN TO CALLING PROGRAM

FIG. 7A

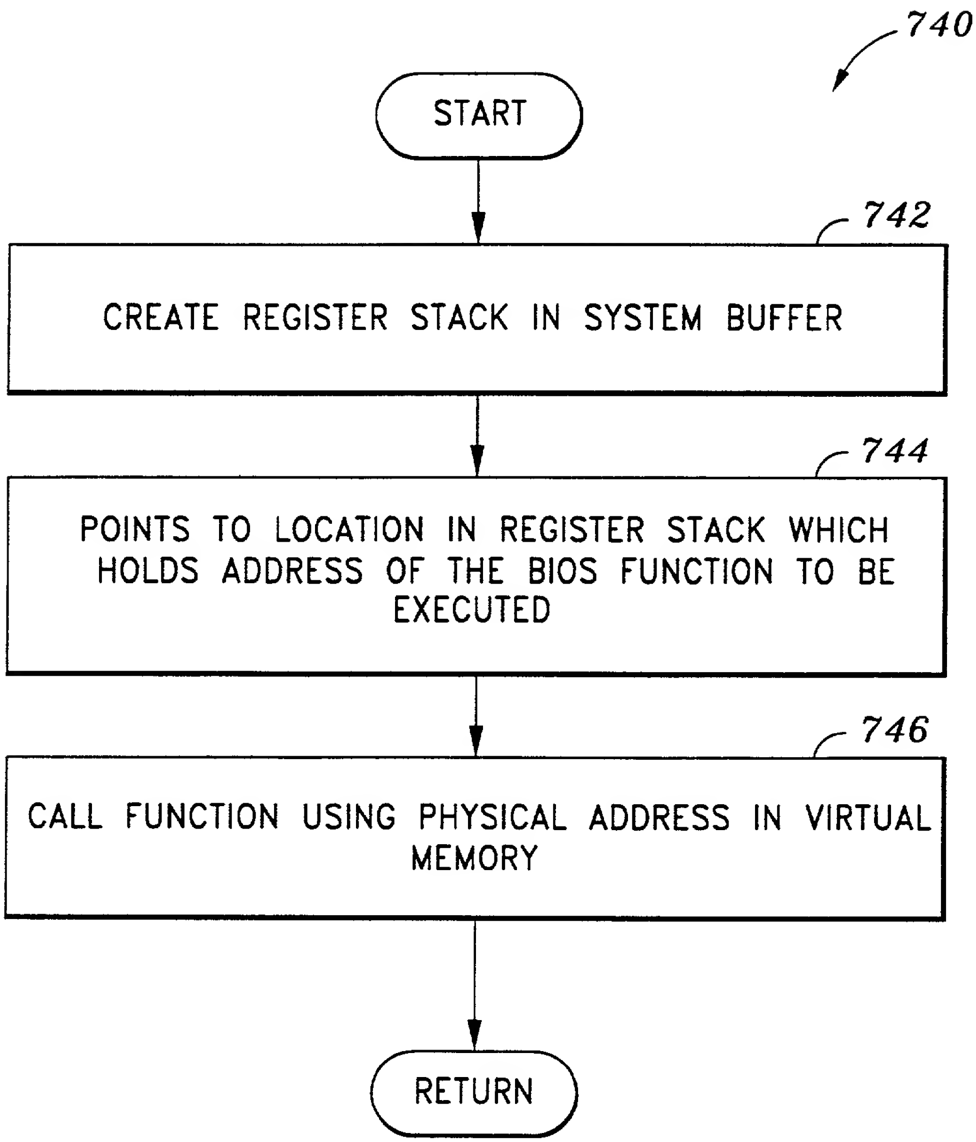

740
START
742
CREATE REGISTER STACK IN SYSTEM BUFFER
744
POINTS TO LOCATION IN REGISTER STACK WHICH HOLDS ADDRESS OF THE BIOS FUNCTION TO BE EXECUTED
746
CALL FUNCTION USING PHYSICAL ADDRESS IN VIRTUAL MEMORY
RETURN

FIG. 7B

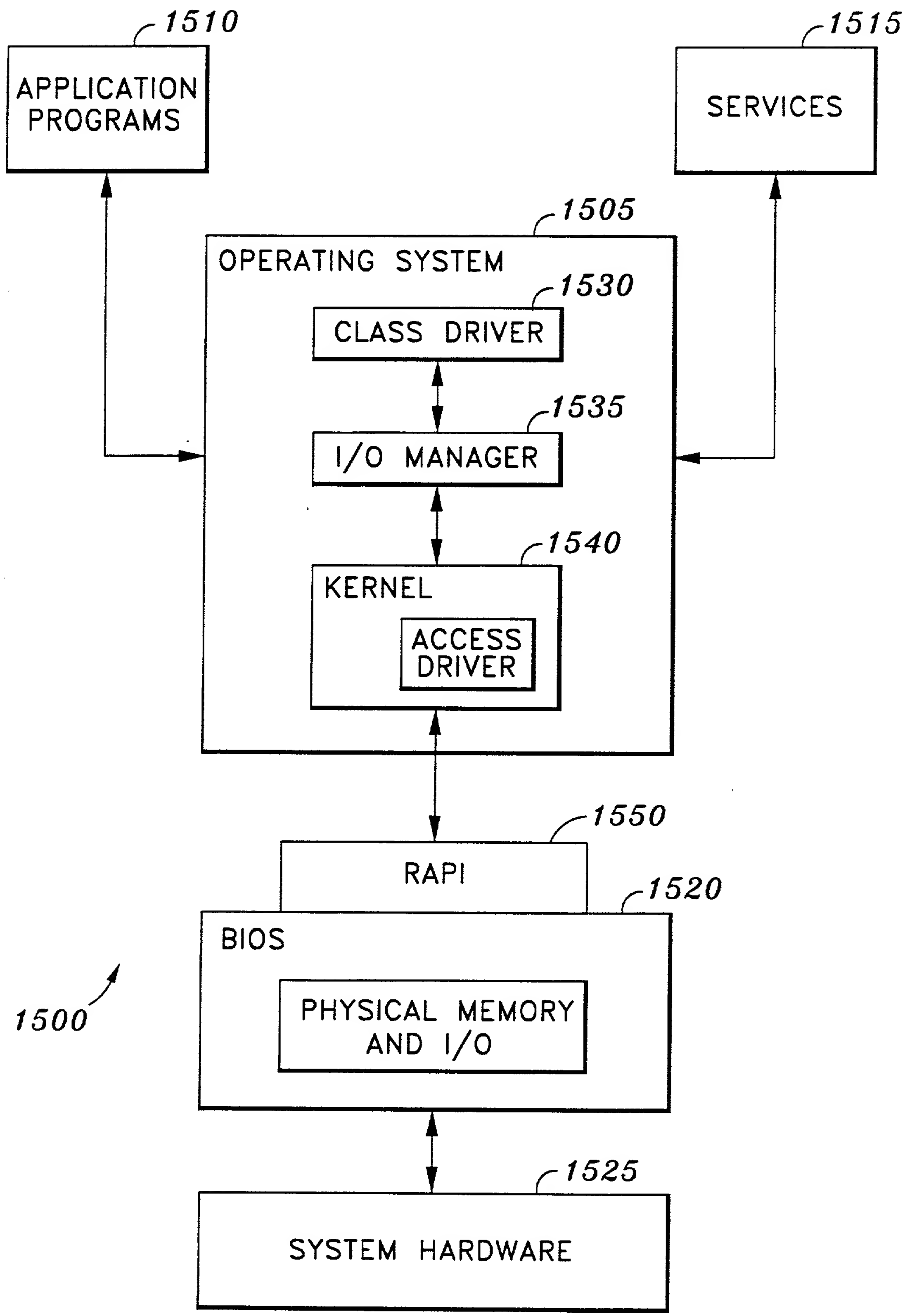
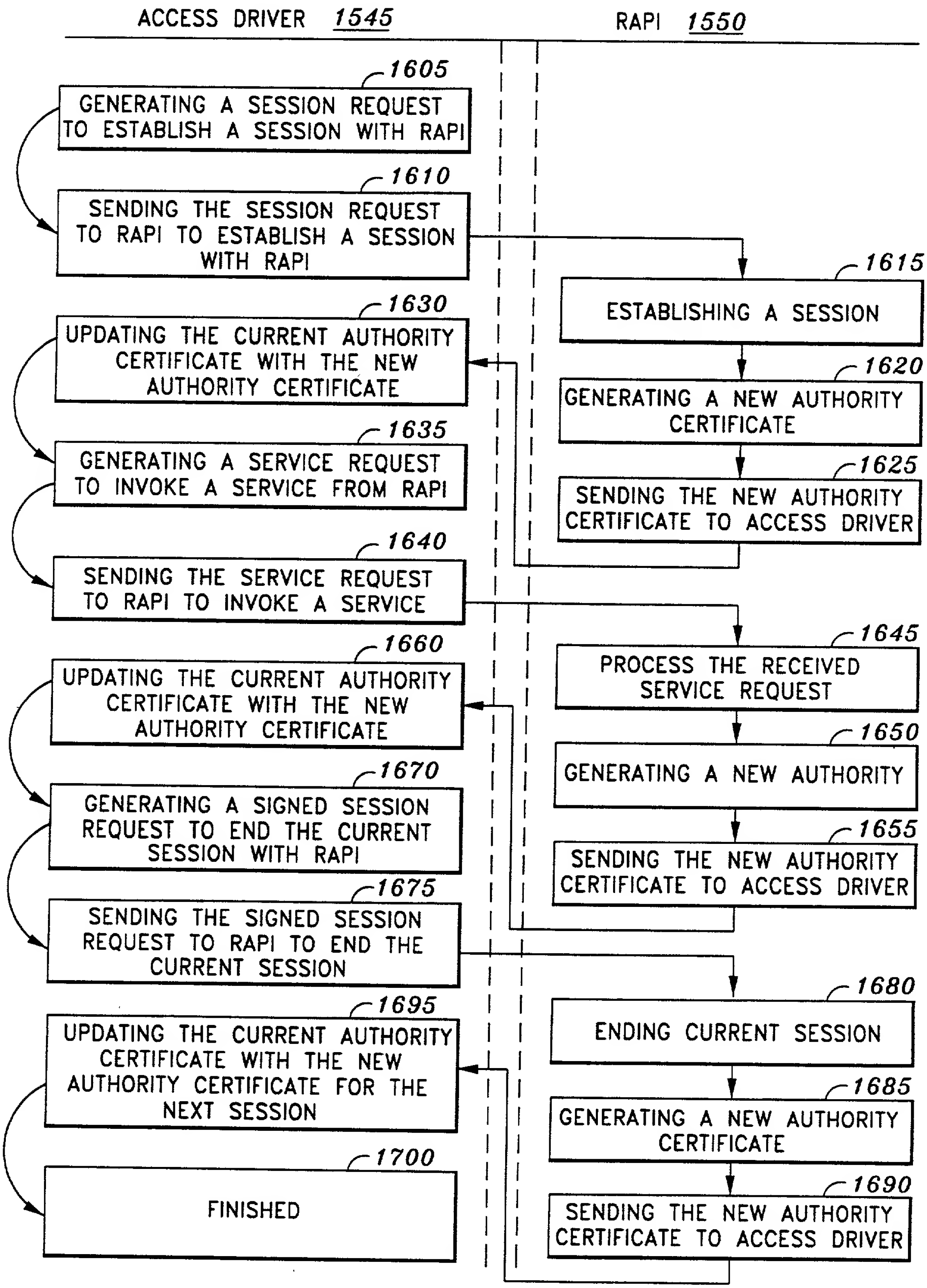

1510
APPLICATION PROGRAMS
1515
SERVICES
1505
OPERATING SYSTEM
1530
CLASS DRIVER
1535
I/O MANAGER
1540
KERNEL
ACCESS DRIVER
1550
RAPI
1520
BIOS
PHYSICAL MEMORY AND I/O
1500
1525
SYSTEM HARDWARE

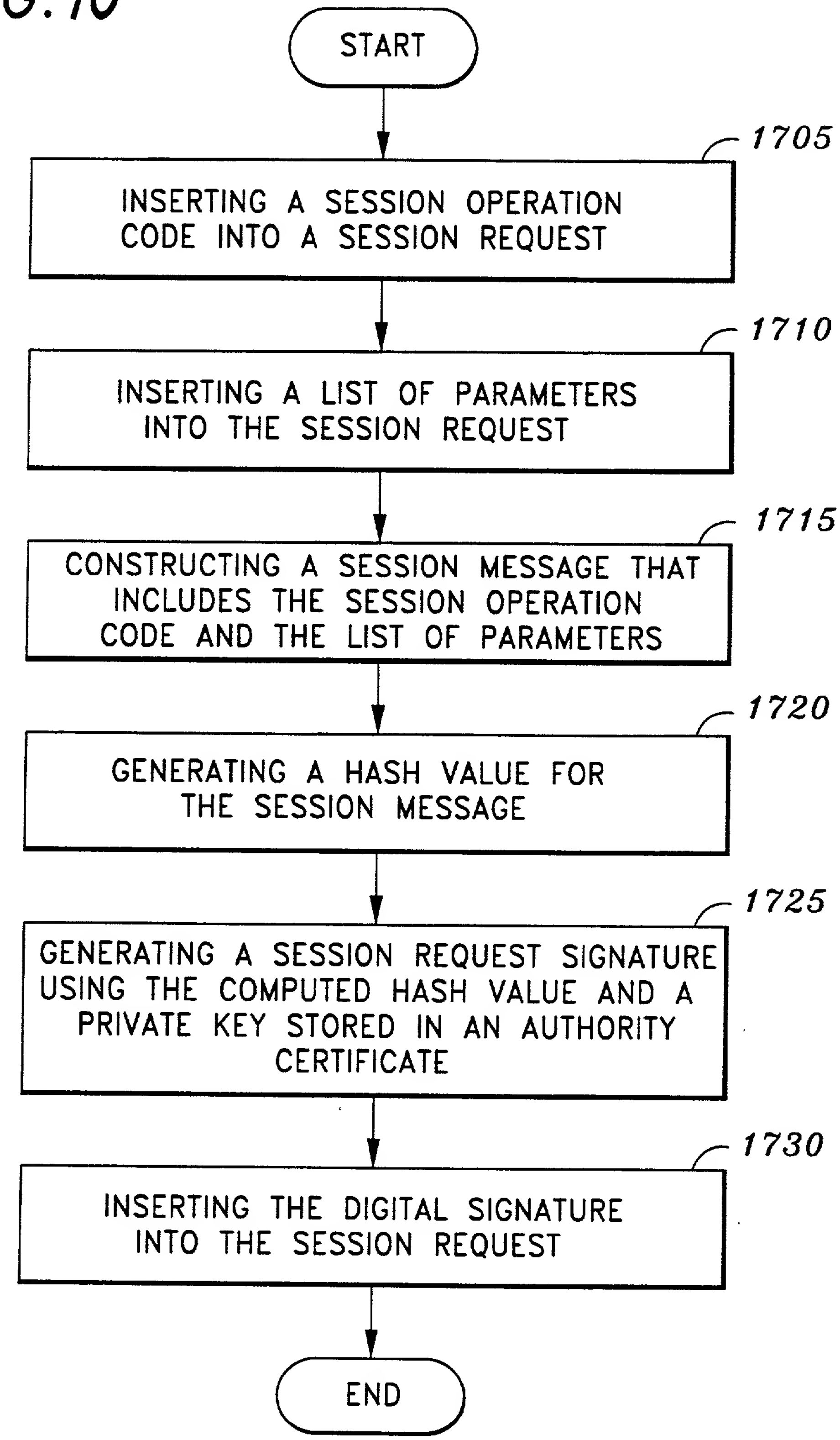
FIG. 8

FIG. 9



ACCESS DRIVER 1545
RAPI 1550
1605
GENERATING A SESSION REQUEST TO ESTABLISH A SESSION WITH RAPI
1610
SENDING THE SESSION REQUEST TO RAPI TO ESTABLISH A SESSION WITH RAPI
1615
ESTABLISHING A SESSION
1620
GENERATING A NEW AUTHORITY CERTIFICATE
1625
SENDING THE NEW AUTHORITY CERTIFICATE TO ACCESS DRIVER
1630
UPDATING THE CURRENT AUTHORITY CERTIFICATE WITH THE NEW AUTHORITY CERTIFICATE
1635
GENERATING A SERVICE REQUEST TO INVOKE A SERVICE FROM RAPI
1640
SENDING THE SERVICE REQUEST TO RAPI TO INVOKE A SERVICE
1645
PROCESS THE RECEIVED SERVICE REQUEST
1650
GENERATING A NEW AUTHORITY
1655
SENDING THE NEW AUTHORITY CERTIFICATE TO ACCESS DRIVER
1660
UPDATING THE CURRENT AUTHORITY CERTIFICATE WITH THE NEW AUTHORITY CERTIFICATE
1670
GENERATING A SIGNED SESSION REQUEST TO END THE CURRENT SESSION WITH RAPI
1675
SENDING THE SIGNED SESSION REQUEST TO RAPI TO END THE CURRENT SESSION
1680
ENDING CURRENT SESSION
1685
GENERATING A NEW AUTHORITY CERTIFICATE
1690
SENDING THE NEW AUTHORITY CERTIFICATE TO ACCESS DRIVER
1695
UPDATING THE CURRENT AUTHORITY CERTIFICATE WITH THE NEW AUTHORITY CERTIFICATE FOR THE NEXT SESSION
1700
FINISHED

FIG. 10



START
1705
INSERTING A SESSION OPERATION CODE INTO A SESSION REQUEST
1710
INSERTING A LIST OF PARAMETERS INTO THE SESSION REQUEST
1715
CONSTRUCTING A SESSION MESSAGE THAT INCLUDES THE SESSION OPERATION CODE AND THE LIST OF PARAMETERS
1720
GENERATING A HASH VALUE FOR THE SESSION MESSAGE
1725
GENERATING A SESSION REQUEST SIGNATURE USING THE COMPUTED HASH VALUE AND A PRIVATE KEY STORED IN AN AUTHORITY CERTIFICATE
1730
INSERTING THE DIGITAL SIGNATURE INTO THE SESSION REQUEST
END

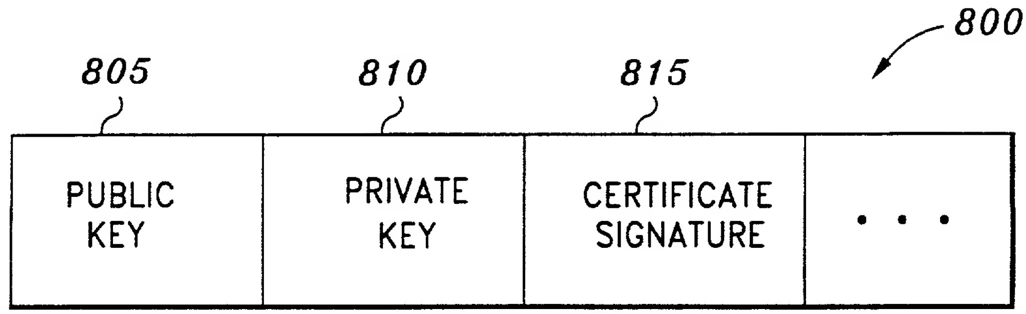


FIG. 11

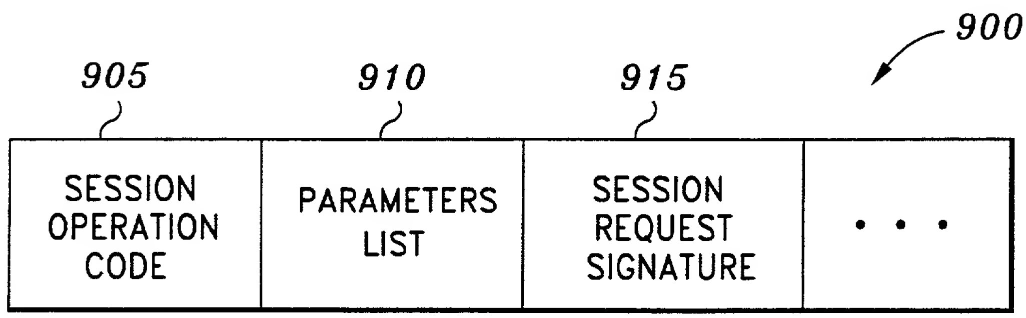


FIG. 12

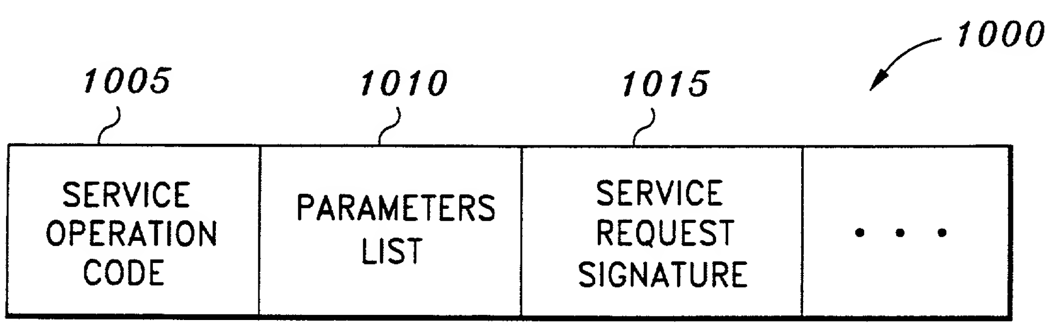


FIG. 13

FIG. 14

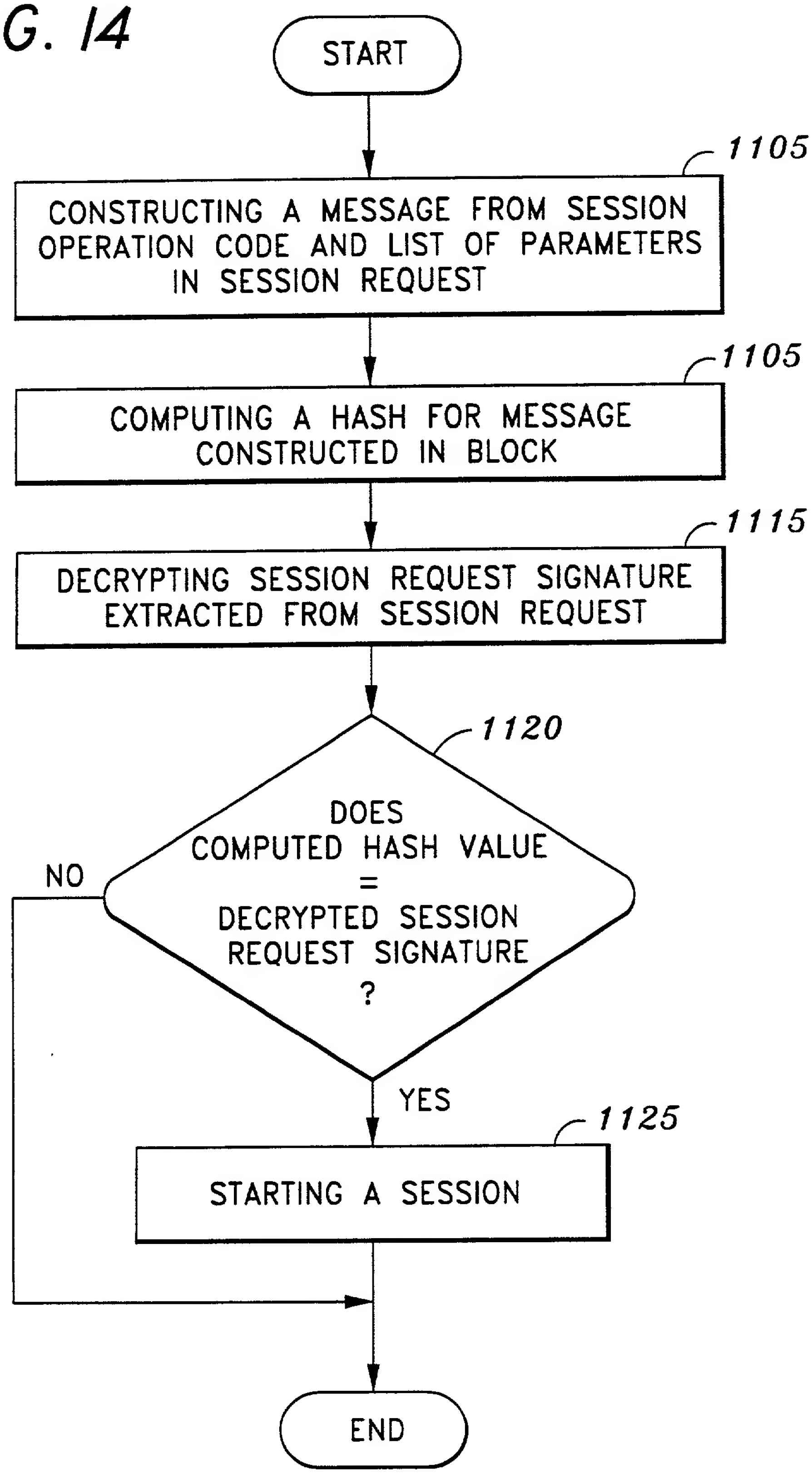

START
1105
CONSTRUCTING A MESSAGE FROM SESSION OPERATION CODE AND LIST OF PARAMETERS IN SESSION REQUEST
1105
COMPUTING A HASH FOR MESSAGE CONSTRUCTED IN BLOCK
1115
DECRYPTING SESSION REQUEST SIGNATURE EXTRACTED FROM SESSION REQUEST
1120
DOES COMPUTED HASH VALUE = DECRYPTED SESSION REQUEST SIGNATURE ?
NO
YES
1125
STARTING A SESSION
END

FIG. 15

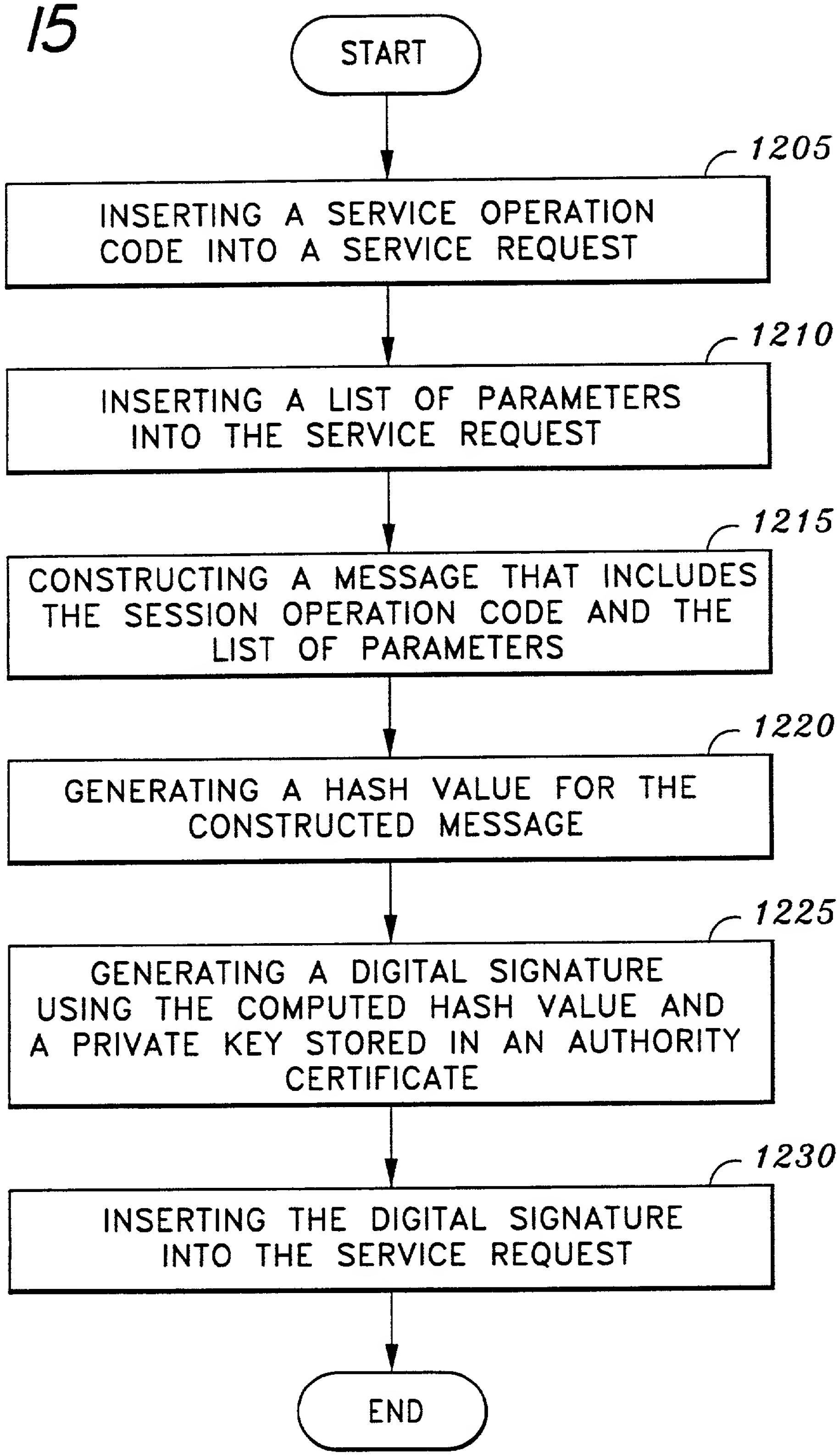

START
1205
INSERTING A SERVICE OPERATION CODE INTO A SERVICE REQUEST
1210
INSERTING A LIST OF PARAMETERS INTO THE SERVICE REQUEST
1215
CONSTRUCTING A MESSAGE THAT INCLUDES THE SESSION OPERATION CODE AND THE LIST OF PARAMETERS
1220
GENERATING A HASH VALUE FOR THE CONSTRUCTED MESSAGE
1225
GENERATING A DIGITAL SIGNATURE USING THE COMPUTED HASH VALUE AND A PRIVATE KEY STORED IN AN AUTHORITY CERTIFICATE
1230
INSERTING THE DIGITAL SIGNATURE INTO THE SERVICE REQUEST
END

FIG. 16

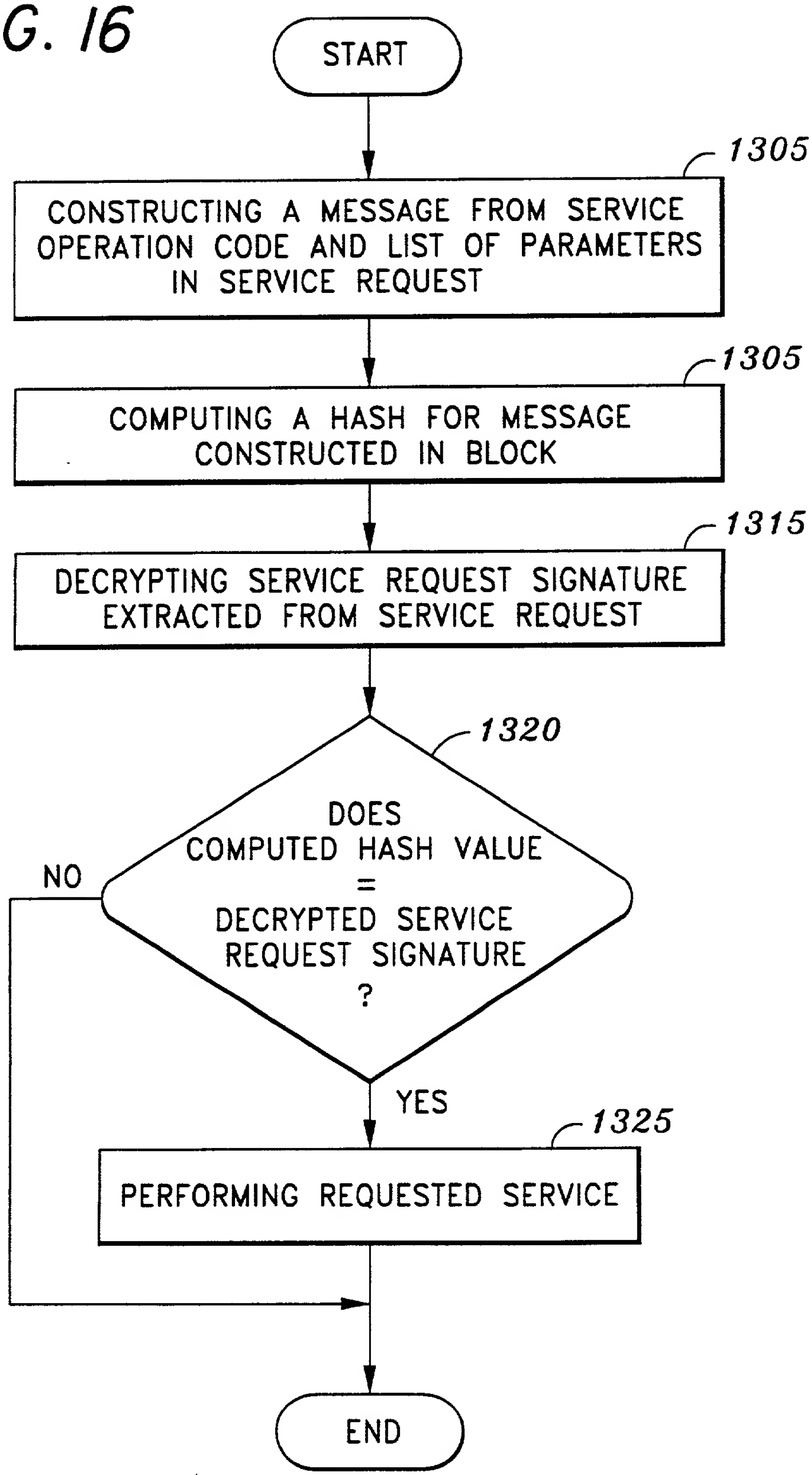
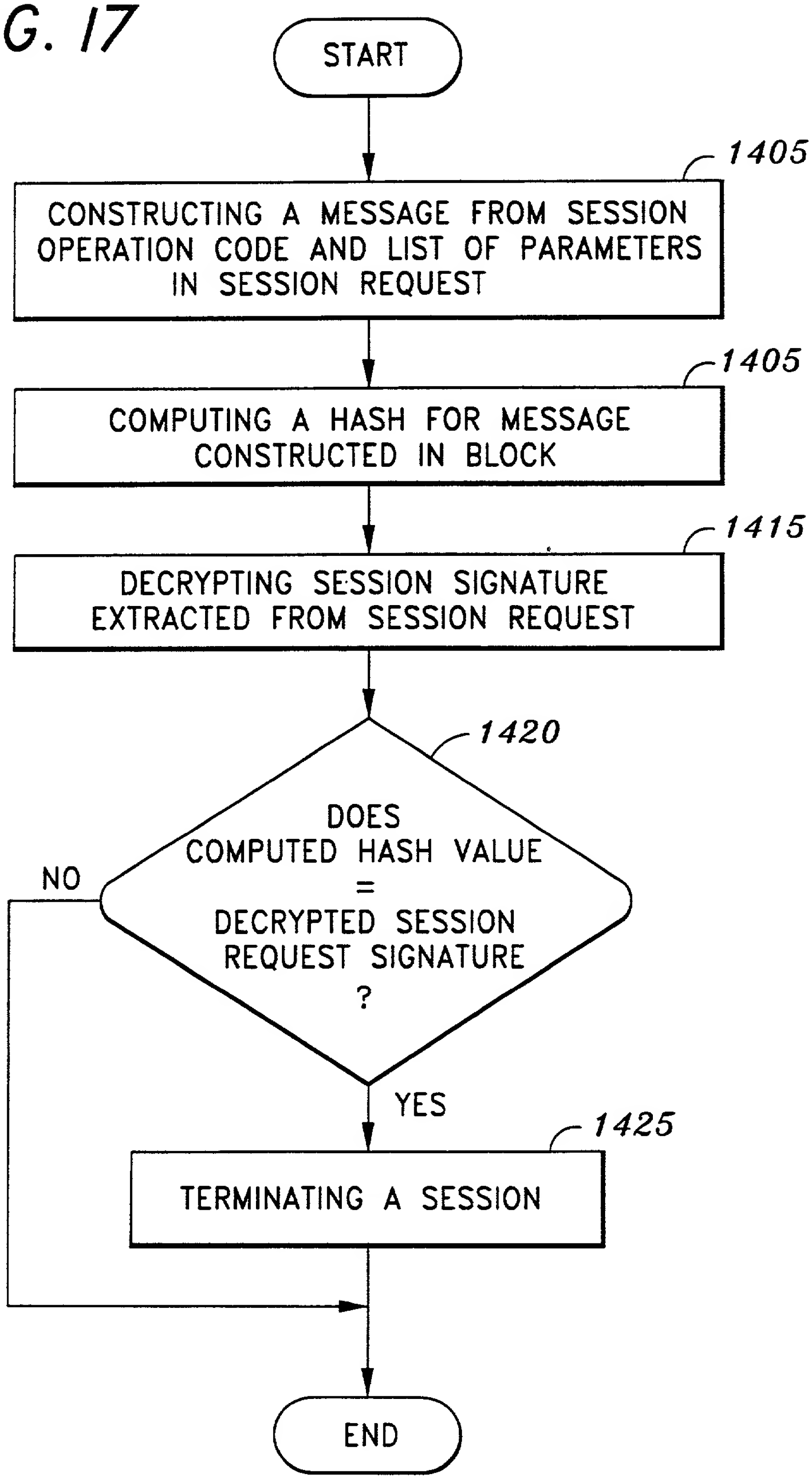
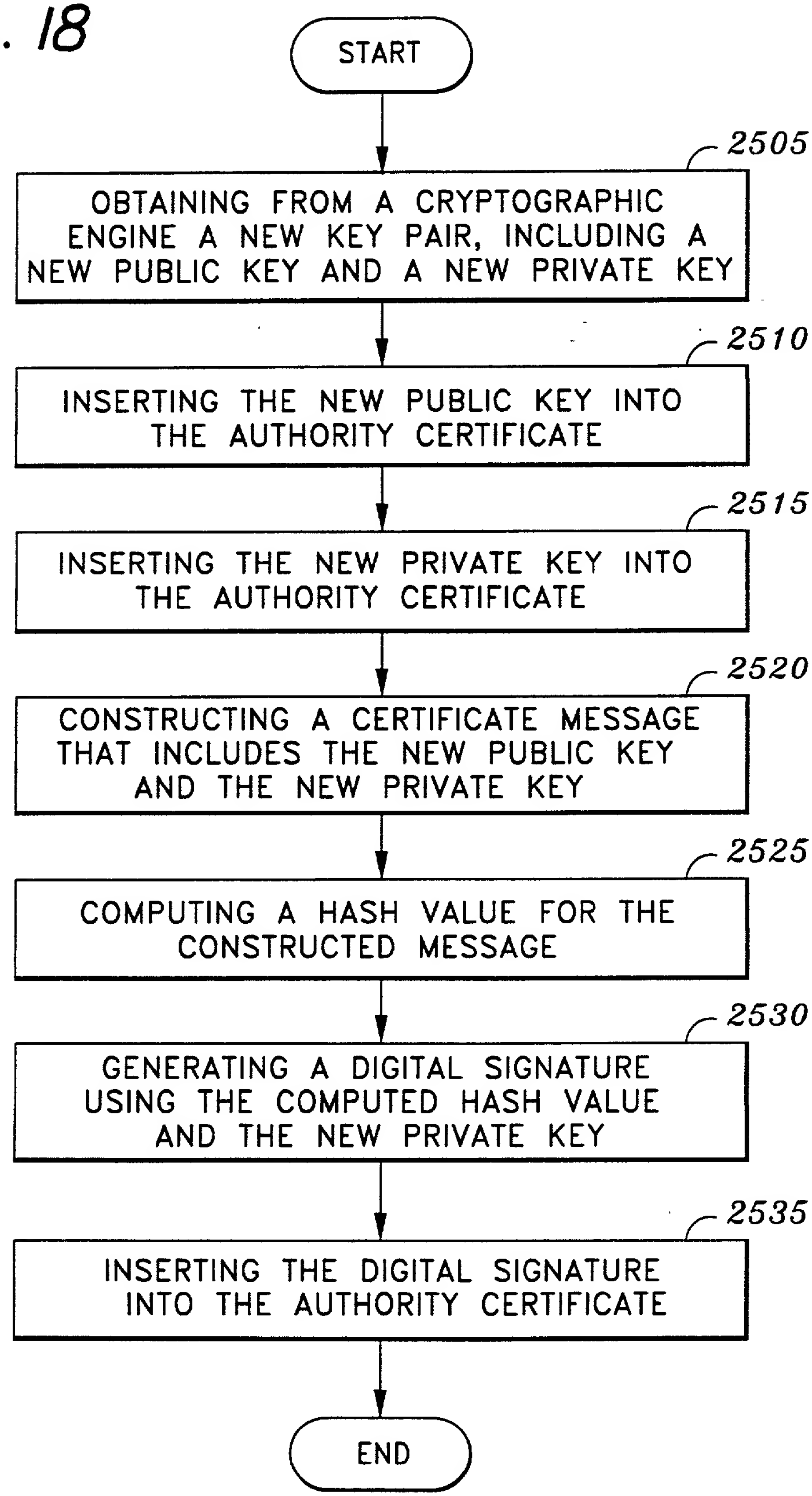

START
1305
CONSTRUCTING A MESSAGE FROM SERVICE OPERATION CODE AND LIST OF PARAMETERS IN SERVICE REQUEST
1305
COMPUTING A HASH FOR MESSAGE CONSTRUCTED IN BLOCK
1315
DECRYPTING SERVICE REQUEST SIGNATURE EXTRACTED FROM SERVICE REQUEST
1320
DOES COMPUTED HASH VALUE = DECRYPTED SERVICE REQUEST SIGNATURE ?
NO
YES
1325
PERFORMING REQUESTED SERVICE
END

FIG. 17



START
1405
CONSTRUCTING A MESSAGE FROM SESSION OPERATION CODE AND LIST OF PARAMETERS IN SESSION REQUEST
1405
COMPUTING A HASH FOR MESSAGE CONSTRUCTED IN BLOCK
1415
DECRYPTING SESSION SIGNATURE EXTRACTED FROM SESSION REQUEST
1420
DOES COMPUTED HASH VALUE = DECRYPTED SESSION REQUEST SIGNATURE ?
NO
YES
1425
TERMINATING A SESSION
END

FIG. 18


START
2505
OBTAINING FROM A CRYPTOGRAPHIC ENGINE A NEW KEY PAIR, INCLUDING A NEW PUBLIC KEY AND A NEW PRIVATE KEY
2510
INSERTING THE NEW PUBLIC KEY INTO THE AUTHORITY CERTIFICATE
2515
INSERTING THE NEW PRIVATE KEY INTO THE AUTHORITY CERTIFICATE
2520
CONSTRUCTING A CERTIFICATE MESSAGE THAT INCLUDES THE NEW PUBLIC KEY AND THE NEW PRIVATE KEY
2525
COMPUTING A HASH VALUE FOR THE CONSTRUCTED MESSAGE
2530
GENERATING A DIGITAL SIGNATURE USING THE COMPUTED HASH VALUE AND THE NEW PRIVATE KEY
2535
INSERTING THE DIGITAL SIGNATURE INTO THE AUTHORITY CERTIFICATE
END

Attorney's Docket No.: 002603.P002X

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below, next to my name.

I believe I am the original, first, and sole inventor (if only one name is listed below) or any original, first, and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled

SYSTEM AND METHOD FOR SECURELY UTILIZING BASIC INPUT AND OUTPUT SYSTEM (BIOS) SERVICES

the specification of which ☐ is attached hereto.
☒ was filed on June 18, 1999 as
United States Application Number 09/336,889
or PCT International Application Number ______
and was amended on ______
(if applicable)

I hereby state that I have reviewed and understand the contents of the above-identified specification, including the claim(s), as amended by any amendment referred to above. I do not know and do not believe that the claimed invention was ever known or used in the United States of America before my invention thereof, or patented or described in any printed publication in any country before my invention thereof or more than one year prior to this application, that the same was not in public use or on sale in the United States of America more than one year prior to this application, and that the invention has not been patented or made the subject of an inventor's certificate issued before the date of this application in any country foreign to the United States of America on an application filed by me or my legal representatives or assigns more than twelve months (for a utility patent application) or six months (for a design patent application) prior to this application.

I acknowledge the duty to disclose all information known to me to be material to patentability as defined in Title 37, Code of Federal Regulations, Section 1.56.

I hereby claim foreign priority benefits under Title 35, United States Code, Section 119(a)-(d), of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed:

Prior Foreign Application(s):

APPLICATION NUMBER	COUNTRY (OR INDICATE IF PCT)	DATE OF FILING (day, month, year)	PRIORITY CLAIMED UNDER 37 USC 119
			☐ No ☐ Yes
			☐ No ☐ Yes
			☐ No ☐ Yes

I hereby claim the benefit under Title 35, United States Code, Section 119(e) of any United States provisional application(s) listed below:

APPLICATION NUMBER	FILING DATE

I hereby claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code, Section 112, I acknowledge the duty to disclose all information known to me to be material to patentability as defined in Title 37, Code of Federal Regulations, Section 1.56 which became available between the filing date of the prior application and the national or PCT international filing date of this application:

APPLICATION NUMBER	FILING DATE	STATUS (ISSUED, PENDING, ABANDONED)

I hereby appoint BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP, a firm including: William E. Alford, Reg.37,764; Farzad E. Amini, Reg. No. 42,261; Aloysius T. C. AuYeung, Reg. No. 35,432; William Thomas Babbitt, Reg. No. 39,591; Carol F. Barry, 41,600; Jordan Michael Becker, Reg. No. 39,602; Bradley J. Bereznak, Reg. No. 33,474; Michael A. Bernadicou, Reg. No. 35,934; Roger W. Blakely, Jr., Reg. No 25,831; Gregory D. Caldwell, Reg. No. 39,926; Lawrence M. Cho, Reg. No. 39,942; Yong S. Choi, Reg. No. 43,324; Thomas M. Coester, Reg. No. 39,637; Roland B. Cortes, Reg. No. 39,152; Barbara Bokanov Courtney, Reg. No. P42,442; William Donald Davis, Reg. No. 38,428; Michael Anthony DeSanctis, Reg. No. 39,957; Daniel M. De Vos, Reg. No. 37,813; Tarek N. Fahmi, Reg. No. P41,402; James Y. Go, Reg. No. 40,621; Richard Leon Gregory, Jr., P42,607; Dinu Gruia, Reg. No. 42,996; Thomas A. Hassing, Reg. No. 36,159; James A. Henry, Reg. No. 41,064; Willmore F. Holbrow III, Reg No. P41,845; George W Hoover II, Reg. No. 32,992; Eric S. Hyman, Reg. No. 30,139; Dag H. Johansen, Reg. No. 36,172; William W. Kidd, Reg. No. 31,772; Tim L. Kitchen, Reg. No. P41,900; Michael J. Mallie, Reg. No. 36,591; Paul A. Mendonsa P42,879; Darren J. Milliken, P42,004; Thinh V. Nguyen, Reg. No. 42,034; Kimberley G. Nobles, Reg. No. 38,255; Michael A. Proksch P43,021; Babak Redjaian, Reg. No. 42,096; James H. Salter, Reg. No. 35,668; William W. Schaal, Reg. No. 39,018; James C. Scheller, Reg. No. 31,195; Maria McCormack Sobrino, Reg. No. 31,639; Stanley W. Sokoloff, Reg. No. 25,128; Allan T. Sponseller, Reg. No. 38,318; Geoffrey T. Staniford, P43,151; Judith A. Szepesi, Reg. No. 39,393; Vincent P. Tassinari, Reg. No. 42,179; Edwin H. Taylor, Reg. No. 25,129; George G. C. Tseng, Reg. No. 41,355; Lester J. Vincent, Reg. No. 31,460; Charles T. J. Weigell, Reg. No. 43,398; Ben J. Yorks, Reg. No. 33,609; and Norman Zafman, Reg. No. 26,250; my attorneys; and Amy M. Armstrong, Reg. No. P42,265; Robert Andrew Diehl, Reg. No. P40,992; and Edwin A. Sloane, Reg. No. 34,728; my patent agents, with offices located at 12400 Wilshire Boulevard, 7th Floor, Los Angeles, California 90025, telephone (714) 557-3800, with full power of substitution and revocation, to prosecute this application and to transact all business in the Patent and Trademark Office connected herewith.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Full Name of Sole/First Inventor (given name, family name) **Leonard J. Galasso**

Inventor's Signature Leonard J. Galasso Date 9/7/99

Residence Rancho Santa Margarita, California USA *(City, State)* Citizenship USA *(Country)*

P. O. Address 21 Tepolito
Rancho Santa Margarita, California 92688 USA

Full Name of Second/Joint Inventor (given name, family name) **Matthew E. Zilmer**

Inventor's Signature ______ Date ______

Residence Upland, California USA Citizenship USA

(City, State) *(Country)*

P. O. Address 2033 N. Second Avenue

Upland, California 91784 USA

Full Name of Third/Joint Inventor (given name, family name) **Quang Phan**

Inventor's Signature Quang Phan Date 9/7/99

Residence Tustin, California USA Citizenship USA

(City, State) *(Country)*

P. O. Address 13281 Saratoga Drive

Tustin, California 92782 USA

Full Name of Fourth/Joint Inventor (given name, family name) ______

Inventor's Signature ______ Date ______

Residence ______ Citizenship ______

(City, State) *(Country)*

P. O. Address ______

Full Name of Fifth/Joint Inventor (given name, family name) ______

Inventor's Signature ______ Date ______

Residence ______ Citizenship ______

(City, State) *(Country)*

P. O. Address ______

Full Name of Second/Joint Inventor (given name, family name) **Matthew E. Zilmer**

Inventor's Signature [signature] Date 8/31/99

Residence Upland, California USA Citizenship USA

(*City, State*) (*Country*)

P. O. Address 2033 N. Second Avenue

Upland, California 91784 USA

Full Name of Third/Joint Inventor (given name, family name) **Quang Phan**

Inventor's Signature Date

Residence Tustin, California USA Citizenship USA

(*City, State*) (*Country*)

P. O. Address 13281 Saratoga Drive

Tustin, California 92782 USA

Full Name of Fourth/Joint Inventor (given name, family name)

Inventor's Signature Date

Residence Citizenship

(*City, State*) (*Country*)

P. O. Address

Full Name of Fifth/Joint Inventor (given name, family name)

Inventor's Signature Date

Residence Citizenship

(*City, State*) (*Country*)

P. O. Address

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of:

LEONARD J. GALASSO;
MATTHEW E. ZILMER;
QUANG PHAN

Application No.: 09/336,889

Filed: June 18, 1999

For: **SYSTEM AND METHOD FOR SECURELY UTILIZING BASIC INPUT AND OUTPUT (BIOS) SYSTEM**

REVOCATION AND POWER OF ATTORNEY

Assistant Commissioner for Patents
Washington, D.C. 20231

Dear Sir:

The Applicant of the above-identified Application hereby revokes all previous powers of attorney given in this Application, and appoints the firm of:

I hereby appoint IRELL & MANELLA LLP, a firm including: Paul Backofen, Reg. No. 42,278; Norman E. Brunell, Reg. No. 26,533; Douglas Carsten, Reg. No. 43,534; Gary Frischling, Reg. No. 35,515; Benjamin Hattenbach, Reg. No. 41,820; Andrei Iancu, Reg. No. 41,862; Soyeon Laub, Reg. No. 39,266; Samuel K. Lu, Reg. No. 40,707; Kimberley G. Nobles, Reg. No. 38,255; Lisa Partain, Reg. No. 40,763; Babak Redjaian, Reg. No. 42,096; Flavio Rose, Reg. No. 40,791; David Rosman, Reg. No. 43,059; Peter Wied, Reg. No. 43,264; Sharon Wong, Reg. No. 37,760; and Ben J. Yorks, Reg. No. 33,609; my patent attorneys, with offices located at 840 Newport Center Drive, Suite 400, Newport Beach, CA 92660, telephone (949) 760-0991, with full power of substitution and revocation, to prosecute this application and to transact all business in the Patent and Trademark Office connected herewith.

Please direct all communications concerning this Application to:

Kimberly G. Nobles
IRELL & MANELLA LLP
840 Newport Center Drive, Suite 400
Newport Beach, CA 92660
(949) 760-0991

PHOENIX TECHNOLOGIES LTD.

Date: 4/10/00

By: [signature]
Name: Thomas Rhodes
Title: Associate General Counsel

STATEMENT UNDER 37 CFR 3.73(b)

Applicant/Patent Owner: Leonard J. Galasso et al.

Application No./Patent No.: 09/336,889 Filed/Issue Date: June 18, 1999

Entitled: System And Method For Securely Utilizing Basic Input And Output (BIOS) System

Phoenix Technologies Ltd. a corporation,
(Name of Assignee) (Type of Assignee, e.g., corporation, partnership, university, government agency, etc.)

states that it is:

1. [✓] the assignee of the entire right, title, and interest; or

2. [ ] an assignee of an undivided part interest

in the patent application/patent identified above by virtue of either:

A. [✓] An assignment from the inventor(s) of the patent application/patent identified above. The assignment was recorded in the Patent and Trademark Office at Reel 10230, Frame 61, or for which a copy thereof is attached.

OR

B. [ ] A chain of title from the inventor(s), of the patent application/patent identified above, to the current assignee as shown below:

1. From:________ To:________
The document was recorded in the Patent and Trademark Office at Reel ______, Frame ______, or for which a copy thereof is attached.

2. From:________ To:________
The document was recorded in the Patent and Trademark Office at Reel ______, Frame ______, or for which a copy thereof is attached.

3. From:________ To:________
The document was recorded in the Patent and Trademark Office at Reel ______, Frame ______, or for which a copy thereof is attached.

[ ] Additional documents in the chain of title are listed on a supplemental sheet.

[ ] Copies of assignments or other documents in the chain of title are attached.
[**NOTE**: A separate copy (*i.e.*, the original assignment document or a true copy of the original document) must be submitted to Assignment Division in accordance with 37 CFR Part 3, if the assignment is to be recorded in the records of the PTO. See MPEP 302-302.8]

The undersigned (whose title is supplied below) is empowered to sign this statement on behalf of the assignee.

May 24, 2000
Date

Kimberley G. Nobles
Signature

Kimberley G. Nobles
Typed or printed name

Attorney
Title

Appendix A

BIOS Power Management Service for Windows Nt 4.0

Introduction

This document provides the interface for the 32-Bit BIOS Power Management Service Interface (BPMSI). This interface is used by the Windows NT Power Management Kernel driver for providing power management services to APM-aware applications.

Terms

APM BIOS

System BIOS which provides power management functions adhering to the APM specification (currently at revision level 1.2)

Kernel Mode

A privileged processor mode in which the NT system code runs. A kernel mode thread has access to all I/O and system memory

Power Management Kernel Driver (PM Driver)

Provides access to the BIOS ROM, BIOS Data Area and the 32-Bit Services for Services.

Power Management Service (PM Service)

Provides power management services for applications and other drivers. Translates their requests into queries to the BIOS Power Management Service Interface using the interface provided by the Power Management Kernel Driver (PM Driver).

PowerPAL

Phoenix's APM BIOS extensions which provide application level access to system and device power management features

System Idle

State where PM service detected minimum processing at the application level. In this state, only threads run on idle priority are executed and they will be preempted by any thread running in a higher priority class.

User Mode

A non-privileged processor mode in which application code runs. An user-mode thread does not have access to I/O and system memory.

Architecture Overview

The extensions to the BIOS for support of Windows NT power management consist of two parts:

(a) Modifications to the BIOS32 Service Directory. An entry was added which allows the PM Driver to find the entry point for the BIOS Power Management Service Interface.
(b) New 32-Bit BIOS Power Management Service, which mimics the APM 32-bit interface with some differences, which are noted below. The new interface provides a 0:32 calling method which is more secure and more Windows NT friendly.

When the PM Kernel wishes to use the BIOS services, it must perform the following steps:

1. Find the BIOS32 Service Directory header
2. Call the BIOS32 Service Directory calling interface, specifying the 32-Bit BIOS Power Management Service Interface entry point.
3. Call the BIOS 32-Bit BIOS Power Management Service Interface entry point, asking for APM Connect.

BIOS32 Service Directory Modifications

The 32-bit BIOS Service Directory is an existing structure within the Phoenix BIOS which allows a 32-bit protected mode application or operating system to find the entry point for a particular 32-bit service. This specification defines a new standard 32-bit BIOS Service.

The BIOS32 Service Directory consists of a fixed structure which can be detected by the PM driver and a single function which returns the address for a particular service.

The BIOS32 Service Directory Header

A BIOS which implements the BIOS32 Service Directory must embed a specific, contiguous 16-byte pattern somewhere in the physical address range 0E0000h - 0FFFFFh. The pattern must be paragraph aligned (i.e., it must start on a 16-byte boundary). This pattern is known as the BIOS32 Service Directory Header.

The Header is comprised of six distinct fields. The following table describes each field.

Offset	Size	Description
0	4 bytes	The ASCII signature "_32_" or 0x5F33325F
4	4 bytes	The entry point for the BIOS32 Service Directory calling interface. This is a 32-bit linear (i.e., not segment:offset) physical address
8	1 byte	The revision level of the BIOS32 Service Directory header and calling interface. The current revision is 0.
9	1 byte	The length of the BIOS32 Service Directory header. Measured in paragraphs (16 bytes). The current length is 1.
10	1 byte	The BIOS32 Service Directory header byte-wide, byte-sum checksum. When totaled, all of the bytes in the header should ad up to 0.
11	5 bytes	Reserved. Should be 0.

Clients of the BIOS32 Service Directory should first determine its existence by locating the Header. This is done by scanning 0E0000h to 0FFFF0h in paragraph increments and looking for a signature match ("_32_") in the first 4 bytes of each paragraph. When, and if, the signature is detected the client should perform a checksum of all bytes in the Header. (The Header length, in paragraphs, is found at offset 9h.) All bytes in the Header should ADD together with a result of 0h. If the checksum is valid then the 32-bit entry point

field can be used as the address for the BIOS32 Service Directory Calling Interface. If the Header is not found then the BIOS32 Service Directory does not exist on the platform.

The BIOS32 Service Directory entry point, and its associated code and data, maybe located anywhere within the 4Gb physical address space. However, it is guaranteed to be physically contiguous (i.e., it will be delivered in ROM or FLASH space) and to fit within two pages (i.e., it will not span three pages).

The BIOS Service Directory

To get the entry point to the BIOS 32-Bit Power Management Service Interface, the PM Driver must call the BIOS Service Directory with the following parameters:

IN:	EAX	"NTPM" or 0x4E54504D
	EBX	0x00000000
OUT:	AL	Error code: 0x00 = None, 0x81 = Service does not exist
	EBX	Base address of the 32-Bit Power Management Service code
	ECX	Length of the 32-bit Power Management Service code (from EBX)
	EDX	Offset (from EBX) of the 32-bit Power Management Service code entry point.

The entry point of the 32-bit Power Management Service code entry point is FAR (that is, requiring both segment and offset to be pushed on the stack.)

CS: The base address must be less than or equal to the (4KB) page address of the page that contains the entry point. For example, if the entry point is 0FFF81234h, then the base address must be less than or equal to 0FFF81000h. The limit must be such that the base address plus the limit generate an address that is greater than or equal to the last address on the (4KB) page which follows the page containing the entry point. For example, if the entry point is 0FFF81234h then the base address plus the limit must be greater than or equal to 0FFF82FFFh. Simply stated, the base address and the limit must "encompass" both the page that contains the entry point and the following page.

The segment type must be 100b (code, execute only) or 101b (code, execute/read). However, the implementers of the Service Directory cannot assume read access to the CS code segment. The system bit must be 1 (non-system segment). It is recommended that the Descriptor Privilege Level (DPL) be 0. (The CS descriptor DPL becomes the Current Privilege Level, or CPL). If the CPL is not 0, then the OS must provide trapping and virtualization services for ring 0 privileged instructions (such as those that access CRx). Note also the dependency of this field on the IOPL field in EFLAGS (see Section0).

The Default Size bit must be 1 (32 bits).

DS: The base address must be equal to the CS base address. The limit must be greater than or equal to the CS limit.

The segment type must be 000b (data, read only) or 001 (data, read/write). However, the implementers of the Service Directory cannot assume write access to the DS data segment. The system bit must be 1 (non-system segment). The Descriptor Privilege Level (DPL) must be greater than or equal to CPL (see the DPL field in Section 0).

SS: The segment type must be 011b (data, read/write, expand-down) or 001b data, read/write, expand-up). The system bit must be 1 (non-system segment). The Descriptor Privilege Level (DPL) must be equal to the CPL (see the DPL field in Section 0). The Default Size bit must be 1 (32 bits). The Granularity bit must be 1 (4Kb).

Note that the above settings ensure a stack size of at least 4kb. It is the caller's responsibility to ensure that there is at least 1kb of unused stack available.

Paging: Paging may or may not be enabled. If paging is enabled, then the address space that is described by the CS and DS selectors must be linearly contiguous. That is, the original physical contiguity of the Calling Interface as found in ROM or FLASH must be preserved. (The Calling Interface code and data is written to be position-independent and EIP-relative).

IOPL: In order for the Calling Interface to execute I/O instructions, the I/O Privilege Level (IOPL) field in EFLAGS must be greater than or equal to the CPL (see the DPL field in Section 0).

Other: The BIOS Data Area, Extended BIOS Data Area and fixed-location ROM data tables cannot be assumed to be available for use by the executing code because of paging. The BDA may be accessed using the pointer provided by the caller.

32-Bit BIOS Power Management Service Interface

In general, the 32-Bit BIOS Power Management Interface provides the same functionality as the 32-bit APM entry point. There are two areas of difference: the way in which calling parameters are passed and the way in which certain APM connect functions are supported.

Calling Parameters

The APM 32-Bit Interface passes all parameters in CPU registers. The BIOS Power Management Service Interface passes the parameters on the stack. The equivalent C-style declaration would be:

```
typedef struct
{
    ULONG        reserved0;      /* 00 */
    ULONG        pBDA;           /* 04 */
    ULONG        regFlags;       /* 08 */
    ULONG        reserved1;      /* 0C */
    ULONG        reserved2;      /* 10 */
    ULONG        reserved3;      /* 14 */
    ULONG        reserved4;      /* 18 */
    ULONG        reserved5;      /* 1C */
    ULONG        reserved6;      /* 20 */
    ULONG        reserved7;      /* 24 */
    ULONG        reserved8;      /* 28 */
    ULONG        reserved9;      /* 2C */
    ULONG        regEBP;         /* 30 */
    ULONG        regEDI;         /* 34 */
    ULONG        regESI;         /* 38 */
    ULONG        regEDX;         /* 3C */
    ULONG        regECX;         /* 40 */
    ULONG        regEBX;         /* 44 */
    ULONG        regEAX;         /* 48 */
} regStruct;

unsigned char BPMSI(regStruct* parameters);
```

The actual APM function and its behavior will be determined by the value written in the corresponding register fields. The pBDA field is a pointer to the virtual address where the kernel driver has mapped the BIOS Data Area so that it can be accessed by the service entrance.

If an error occurs, bit 0 of regFlags will be 1 and the error code will be in bits 8-15 of regEAX, otherwise it will be 0 and bits 8-15 of regEAX will be 0. The error codes are identical to those found in the APM 1.2 specification.

Software Requirements

Development Software Requirements

The following software development tools are required to build the PM service:

- MASM 6.11c assembler from Microsoft
- Phoenix PhDebug for debugging the code

Software Requirements

PM service requires to run on the following environment:

- Windows NT 4.0
- System with Phoenix NoteBIOS supporting the NT BIOS Interface to the PowerPAL
- PM driver installed in the system

Porting Changes

The following section describes the files which have been modified in the Core, Miser and the chipset code. It also describes the changes which need to be made in order to port a normal BIOS to include the NT support.

Appendix B

PhoenixPhlash NT

Flash ROM Programming Utility for Windows NT

The PhoenixAD driver referred to in Appendix B is the Access Driver 46.

1.0 Overview

The PhoenixPhlash flash utility will be used to program BIOS images into flash ROMs in AT compatible systems. The utility will consist of the following files:

PHLASHNT.EXE	- used to program the flash ROM
PLATFORM.DLL	- used to perform platform dependent functions
BIOS.ROM	- actual BIOS image to be programmed into flash ROM

This specification provides a detailed description of the functionality for the PHLASHNT.EXE program. Because PLATFORM.DLL and BIOS.ROM are platform specific, only the general format of these two files is covered in this document.

PhoenixPhlash will be executed as a Win32 console application.

High priority is being placed on flexibility, adaptability and supportability for this design project. As much customization capability as possible will be placed into the PLATFORM.DLL file so that many different platforms and configurations can be supported without modifying PHLASHNT.EXE. PhoenixPhlash will support platforms with a single flash ROM part, as well as platforms with multiple flash ROMs. Flash ROMs from 1Mbit to 4Mbits or greater will be accommodated, including boot block devices and devices with any configuration of multiple flashable regions.

For each supported part, all code specific to the particular flash ROM part (e.g. Intel 28Fxxxx) will be part of the PHLASHNT.EXE module. All code and parameters specific to a platform (e.g. flash enable code and flash ROM address range) will be part of the PLATFORM.DLL module.

PHLASHNT.EXE, the main module of the PhoenixPhlash utility, will contain all code which is platform independent. It will contain user interface code, code to load and verify the PLATFORM.DLL file and the platform independent portions of code to program a flash device.

PHLASHNT.EXE will be a Win32 executable file, generated using Microsoft C++ V4.2 or later.

Using CDriver to access hardware

PHLASHNT.EXE uses the CDriver C++ class which works in conjunction with the *PhoenixAD* driver to enable Windows NT user-mode applications to access I/O ports; to access BIOS data and code area; and to execute BIOS32 services. The CDriver class provides a simple and flexible interface between the application program and the *PhoenixAD* driver.

CDriver works in conjunction with the *PhoenixAD* driver to provide the following functions to user-mode application programs.

Access to I/O ports
Execute BIOS32 services

Access the BIOS image
Access BIOS data areas
Read the system Real Time Clock

The **CDriver** class serves as a thin wrapper interface between Windows NT applications and the *PhoenixAd* driver. It encapsulates the interface to the driver and provides flexibility to both applications and the kernel driver designs.

To assure future compatibility, PHLASHNT.EXE does not call the *PhoenixAD* driver directly; instead, it calls the methods in the **CDriver** class.

2.0 Modes of Operation

2.1 Win32 Console

PHLASHNT.EXE will be started in a Windows NT window, followed by optional command line flags (if any).

Command line flags will include (both lower and upper case characters are acceptable):

/A
AUTODETECT OFF - Do not read ID from the part. By default, program verifies that the manufacturer ID and part ID read from the part, matches the ID specified in the PLATFORM.DLL file and when the two IDs differ, the ID read from the part is used. This allows one to use the same PLATFORM.DLL and BIOS.ROM for several different parts without the need to modify either of the two files. When this flag is not set, then it is assumed that the ID is "readable" from the part. When this flag is set, the ID from the part is not used, instead the values specified in the PLATFORM.DLL are used.

/B=*filename*
BINARY FILE - Overrides the default platform specific binary file. This option is required when a full path specification is needed and/or the binary file has a name other than PLATFORM.DLL.

/BU=*filename*
BACKUP - Save the previous version of the BIOS image into file *filename* before erasing. *Filename* is optional; if not specified, the previous image is stored in BIOS.BAK. Because many versions of BIOS use platform dependent features such as shadow memory and de-compression, it is often necessary to use platform dependent code in PLATFORM.DLL to retrieve the BIOS image before it can written to a file.

/C
CMOS UPDATE - Clears the CMOS checksum after Flash is updated. If the AUTO_UPDATE feature is installed in the new BIOS image, the BIOS automatically sets all CMOS fields to their default values on the next boot. If the AUTO_UPDATE feature is not loaded, the BIOS displays the CMOS checksum error message on the next boot and prompts the user to press the F2 key to execute Setup and manually reconfigure the machine.

/CS
CHECKSUM BIOS ROM - Computes checksum on BIOS ROM image. If the checksum is not zero, or if the optional PLATFORM.DLL function CheckSum fails, the program terminates with an error message.

/H
USAGE - Display program name, version, copyright and help screen. /? can also be used for this option.

/I
IMAGE SIZE VERIFICATION - Proceed only if the ROM image file size is the same as the size of the flash part.

/MODE=n
OPERATION - Selects the operating mode for PHLASHNT. The following operating modes are currently supported:

0 Update only the BIOS image (the normal operating mode). In this mode, PHLASHNT replaces the current BIOS image with the new image. The DMI information in the system BIOS is maintained. This is the default mode and is selected if the /MODE command line flag isn't present or if an operating mode isn't specified.

1 Update only the DMI information. In this mode, PHLASHNT writes the strings specified via the DMI command line flags to the Flash. The DMI information in the system BIOS is maintained unless new DMI strings are specified on the command line.

2 Update both the BIOS and DMI information (save system DMI strings). In this mode, PHLASHNT both replaces the current BIOS image and writes the strings specified via the DMI command line flags to Flash. The DMI information in the system BIOS is maintained unless new DMI strings are specified on the command line.

3 Update both the BIOS and DMI information (reset system DMI strings). In this mode, PHLASHNT both replaces the current BIOS image and writes the strings specified via the DMI command line flags to Flash. The DMI information in the system BIOS is replaced with the DMI strings from the new BIOS ROM image and/or new DMI strings specified on the command line.

These options are not displayed by the help (/H) option for security reasons.

/N
NEW (different) - Proceed only for different version of BIOS ROM. If the data structure at BCPSYS, which includes BIOS version and build date & time, is same as the corresponding structure in the BIOS.ROM file then the programs terminates without flashing.

/O
OVERRIDE PLATFORM.DLL OPTIONS - Disable all flags set in PLATFORM.DLL. Without this switch, options set in the PLATFORM.DLL are combined with options specified on the command line. When this switch is used, only command line options are used.

/P
PRODUCTION - Maximize speed of flashing. All user feedback is reduced to a minimum (no sound, or screen update). This is used to reduce the time needed to flash a part in a production environment. Only the final success/failure is reported.

/PN
BIOS PART NUMBER CHECK - Proceed only if the BIOS part number in BIOS.ROM is the same as the part number in the current BIOS.

/PF="list of options"
Command line options to be passed to the platform dependent module PLATFORM.DLL. On some platforms it may be desirable to pass command line options to the platform dependent procedures. This is done via the CmdLine() function. When both the CmdLine() address is non-zero and this command line option is present, then the string immediately following the equal sign will be passed to PLATFORM.DLL (enclose the string in double quotes if the string includes spaces).

/Rn
RETRY - If flashing a block fails, retry n times instead of aborting.
The /Rn option can be used in crisis mode by setting psiRetryCount with the desired retry count in PLATFORM.CPP.

/S
SILENT - Silent operation without audio feedback.

/V
VERIFY - After each block is programmed, data in the flash part address space will be compared to the data in the BIOS.ROM file. Any discrepancies are reported and the program will either re-try programming of the same block or the system will halt (depending on the response to a prompt). Because the check is made after the flash memory was erased, the system will be very unstable and it may not be possible to properly notify the user and recover.

/Z
ZERO BLOCKS - Zero flash blocks before erasing.

filename
BIOS ROM image file name. Any command line option without the leading back-slash will be interpreted as the file name for the BIOS ROM image file. A filename is only required when necessary to specify a full path for the ROM BIOS image and/or the ROM BIOS image file is different than BIOS.ROM.

@filename
Response file. Any of the command line options described above may be placed in a response file. PHLASHNT will read the file and process the options as though they were entered on the command line. The options may be placed on a single line or on separate lines. Each line may be up to 1024 characters in length.

The following command line flags are used to write information to Flash for later retrieval through the Phoenix Desktop Management Interface (DMI). DMI command line flags are ignored if the target BIOS image does not support the DMI interface (doesn't have a DMI BCP structure installed) or the PHLASHNT operation mode is BIOS only (see above).

All flags have the format /Dxx:*String*, where xx is one or two characters identifying the specific DMI string (see below). DMI command line flags are optional; *i.e.*, if a given DMI command line flag isn't specified, the previous contents of the corresponding DMI string buffer aren't modified, unless a default string is specified in PLATFORM.DLL. In this case, PHLASHNT always writes the default string to the corresponding DMI string buffer. If a DMI command flag is specified without the *String* field, the corresponding DMI string buffer is cleared (set to a null string). *String* can only contain printable ASCII characters. *String* must be enclosed in quotes if it contains spaces. The maximum length of each DMI string is platform specific; PHLASHNT returns an error if the passed string is longer than the corresponding target buffer. The following DMI fields are currently supported. These options are not displayed by the help (/H) option for security reasons.

/DSS:*String*	Specifies the system serial number string.
/DMS:*String*	Specifies the system manufacturer's name string.
/DPS:*String*	Specifies the system product (model) identification string.
/DVS:*String*	Specifies the system version string.
/DSM:*String*	Specifies the motherboard serial number string.
/DMM:*String*	Specifies the motherboard manufacturer's name string.
/DPM:*String*	Specifies the motherboard product (model) identification string.
/DVM:*String*	Specifies the motherboard version string.
/DSC:*String*	Specifies the chassis serial number string.
/DMC:*String*	Specifies the chassis manufacturer's name string.
/DPC:*String*	Specifies the chassis product (model) identification string.
/DVC:*String*	Specifies the chassis version string.
/DO1:*String*	Specifies OEM string 1.
	...
/DO*n*:*String*	Specifies OEM string n.

The system and chassis switches are available only with DMI version 2.0.

The older forms of the command line switches, given below, were originally for DMI 1.2 and are kept for compatibility. These are equivalent to /DSM, /DMM, /DPM and /DVM respectively.

/DS:*String*	Specifies the motherboard serial number string.
/DM:*String*	Specifies the motherboard manufacturer's name string.
/DP:*String*	Specifies the motherboard product (model) identification string.

/DV:*String* Specifies the motherboard version string.

Next, the flash program will load the PLATFORM.DLL file and call the platform dependent function EnableFlash() in PLATFORM.DLL to prepare the platform for flashing. PLATFORM.DLL will indicate memory region where the BIOS ROM image is to be loaded in memory and will indicate what memory regions to use for the flash device. Memory regions may be in conventional memory or in extended memory. After the ROM image is loaded in memory device programming begins.

For each block of the flash ROM to be programmed:

1) BeginFlash() is called in PLATFORM.DLL
2) The proper flash algorithm in PHLASHNT.EXE is executed.
3) EndFlash() is called in PLATFORM.DLL.

This process is repeated for each flash block specified in PLATFORM.DLL. This allows for multiple devices on a single platform, multiple blocks within a device and block dependent initialization/termination code for each block. This also allows for automatic saving and restoring of memory regions such as boot blocks.

During flashing, progress information is presented to the user:

1) If production mode is not selected, an appropriate message window will be displayed on the screen, which will include time of day, gas gauge style progress indicator and status line message.

2) Approximately once every second a short beep is sounded.

3) At the start and completion of each step an appropriate code is sent to the debug port.

Video will be updated at least once every second. The sound will be generated approximately every second. Note that in production environment progress update can be disabled.

After flashing is complete, DisableFlash() is executed from the PLATFORM.DLL file. One of two distinct sounds will be generated to indicate success or failure of the flash process. If video is available, an appropriate message window will be displayed. After a short pause system is re-booted.

2.3 Completion Codes

Although the program will proceed through many steps and will be capable of reporting to the user status at each step, only three major stages are identified by sound and keyboard LED codes. The three major stages are:

1) Read and verify PLATFORM.DLL file
2) Perform platform specific initialization
3) Program the part

If the program fails to complete any of the three major stages of the flashing process, program will use distinct sound sequence and keyboard LEDs to inform user at which stage the program failed. At the start of the program CAPS_LOCK, NUM_LOCK and SCROLL_LOCK LEDs on the keyboard will be turned on. The failure at each of the three stages is indicated as follows:

Sound	Keyboard LEDs ON	Description
low buzz + 3 short tones	CAPS, NUM, SCROLL	1. Before reading platform.dll
low buzz + 2 short tones	CAPS, NUM	2. Before platform init
low buzz + 1 short tone	NUM	3. After platform init
1 long tone	none	Successful completion

Stage 1 - Failure occurred before locating PLATFORM.DLL. Most likely due to errors in PLATFORM.DLL file format. System is in stable mode, no changes have been made to the BIOS, no re-boot is needed.

Stage 2 - Failure during platform dependent initialization. System is unstable and re-boot is needed. No changes have been made to the BIOS.

Stage 3 - Failure during programming of the flash memory. System is unstable and the BIOS flash memory is corrupted. System must be restarted with Crisis Diskette.

The program will also perform error detection at each of the steps. Unless explicitly disabled by the PRODUCTION option, as the program progresses it will report on the screen and debug port either the step number of the step currently in progress, or one of the error codes. Code numbers for errors and program steps are further defined in Appendix B2:

When an error is detected before any changes are made to the flash memory part, then program will attempt to notify the user and then will exit PHLASHNT with proper error messages. Once the flash memory has been modified, errors will cause the program to halt.

2.4 Device Dependent Modules

For each type of flash memory supported, there will be a part specific module to perform the following:

1) Identify the part and return the manufacturer ID and the part ID

2) Zero a range of flash memory (set all bits to 0)

3) Erase a range of flash memory (set all bits to 1)

4) Program a range of flash memory

2.4.1 Autodetection

In this module will be the code necessary to read the Manufacturer ID and part ID from the flash memory part. If IDs cannot be determined, zero is returned. The built-in auto-detect module will not be used when the AutoSense() function is provided in the PLATFORM.DLL file; the provided AutoSense() function will be used instead.

2.4.2 Zero

There are several part types which require that the flash memory is set to zero before it can be programmed. In this module will be the code necessary to set memory range to zeros.

2.4.3 Erase

Most flash memory parts require that the flash memory is set to all ones before the part can be programmed. These parts often allow erasure of a full block of flash memory with a single write operation. In this module will be the code necessary to set memory range to ones.

When block descriptors are defined in the PLATFORM.DLL file, descriptors must be set up so that there is at least one descriptor for each "erasable" block in the flash memory. For example in Intel 28F004 flash memory there is one 16kByte BOOT block, two 8kByte PARAMETER blocks, one 96kByte MAIN block and three 128kByte EXTENDED blocks. Each of the seven blocks can be erased with a single write and there must be at least one descriptor for each of the seven blocks.

2.4.4 Program

In this module will be the code necessary to read the data bytes form the BIOS ROM image and program these into the flash part.

2.5 Supported Devices

Initial set of flash memory devices supported by PHLASHNT.EXE will include the parts listed in the table below. For each part type a manufacturer and part ID and part description is listed. As new parts become available it may be necessary to add additional modules to PHLASHNT.EXE so that a new type of flashing algorithm is provided (new AutoDetect(), Zero(), Erase() and Program() functions). If it is possible for the new part to use one of the existing algorithms and only the Manufacturer or Part ID changes, then this can be indicated in the PLATFORM.DLL file and no modification of PHLASHNT.EXE is needed (see section on PartTypes for more detail).

Type	Mfg ID	Part ID	Description
2	0x01	0xA1	AMD 28F256
2	0x01	0x25	AMD 28F512
1	0x01	0xA7	AMD 28F010
1	0x01	0xA2	AMD 28F010A
2	0x01	0x2A	AMD 28F020

2	0x01	0x29	AMD 28F020A
2	0x01	0x20	AMD 29F010
2	0x01	0xA4	AMD 29F040
2	0x01	0x51	AMD 29F200T
10	0x01	0xB0	AMD 29F002T
10	0x01	0x34	AMD 29F002B
10	0x01	0xDC	AMD 29F002BXT
10	0x01	0x5D	AMD 29F002BXB
3	0x1F	0xD5	ATMEL 29C010
8	0x1F	0xDA	ATMEL 29C020
3	0x1F	0x35	ATMEL 29LV010
1	0x1C	0xD0	Mitsubishi 28F101
1	0x89	0xB9	Intel 28F256
1	0x89	0xB8	Intel 28F512
1	0x89	0xB4	Intel 28F010
1	0x89	0xBD	Intel 28F020
4	0x89	0x94	Intel 28F001 BX-T
4	0x89	0x95	Intel 28F001 BX-B
4	0x89	0x7C	Intel 28F002 BX-T
4	0x89	0x7D	Intel 28F002 BX-B
4	0x89	0x74	Intel 28F200 BX-T
4	0x89	0x75	Intel 28F200 BX-B
4	0x89	0x78	Intel 28F004 BX-T
4	0x89	0x79	Intel 28F004 BX-B
4	0x89	0x70	Intel 28F400 BX-T
4	0x89	0x71	Intel 28F400 BX-B
5	0xBF	0x07	SST 29EE010/29LE010
12	0xBF	0x10	SST 29EE020
5	0xBF	0x5D	SST 29EE512
6	0xBF	0x04	SST 28SF040
1	0x20	0x07	ST M28F101
7	0xC2	0x11	MX 28F1000
11	0xC2	0x2A	MX 28F2000

3.0 PLATFORM.DLL Detail

This module contains all platform dependent code and parameters needed to program a flash device on a particular platform.

3.1 File Format

PLATFORM.DLL is a Windows DLL that is produced by compiling PLATFORM.CPP (using Microsoft Visual C++ 4.2 or later). It contains specific platform data and executable code. A sample source code of the PLATFORM.CPP file is included in Appendix B3.

File version for PLATFORM.DLL will start with version "NT 1.00". Versions are specified in the szVersion variable contained in PLATFORM.DLL.

3.2 File Header Format

The PLATFORM.DLL file will have the format described below:

```
                // ====================================
                // Global Variable Declarations
                // ====================================

DWORD    dwFileSize        // ROM image file size
BYTE     bManufactID       // Manufacturer of flash device
BYTE     bPartID           // Part ID of flash device
DWORD    dwFlags           // Option flags
DWORD    dwImageBuf        // Linear address of image buffer
DWORD    dwMfgIDAddr       // Linear address of mfg ID
DWORD    dwPartIDAddr      // Linear address of part ID
BYTE     bRetryCount       // Count for /Rn option (default = 0)
char     szVersion[]       // PLATFORM.DLL version
char     szROMFileName[]   // Name of BIOS image file
DWORD    dwDLLFuncDefine   // Indicates what functions are defined
BYTE     bblockTableSize   // number of blocks in blockTable
BLOCK_DESCRIPTOR blockTable[]
BYTE     bpartTypesSize    // number of flash parts to add
DEVICETABLE partTypes[]
```

dwFileSize Number of bytes in the BIOS.ROM file.

bManufactID Manufacturer ID

bPartID Part ID

dwFlags Option flags. Must be combination of the following values:

FLAG_AUTOSENSEOFF	Don't read ID from the part
FLAG_BACKUP	Backup system BIOS ROM
FLAG_NEWBIOSONLY	Don't flash if 64k at F000:0 same
FLAG_PRODUCTION	Max speed (sound & video off)

FLAG_SILENT	Do not generate any sounds
FLAG_VERIFY	Verify each block after flash
FLAG_PLATFORMCMD	PLATFORM option str present
FLAG_BIOSPARTNUM	Flash only same BIOS part number
FLAG_CHECKSUM	Checksum BIOS.ROM
FLAG_CMOS	Clear CMOS checksum
FLAG_IMAGESIZE	Verify image size matches flash part

dwImageBuf
Address for BIOS.ROM image buffer in extended memory. This field determines the linear address of a buffer where the image will be read into.

This area is also used when the SAVE option is specified. Any blocks to be saved will be using the address range starting at the address dwImageBuffer + dwFileSize.

dwMfgIDAddr
dwPartIDAddr
These two optional fields contain the linear addresses for the flash memory ID bytes. When these fields are zero, the default addresses of E0000h and E0001h will be used.

bRetryCount
Number of times to retry if flashing fails.

szVersion
Version of PLATFORM.DLL

szROMFileName
Reserved must be the string "BIOS.ROM". This field is used to identify and verify format of the PLATFORM.DLL file.

dwDLLFuncDefine
Indicates what platform-specific functions are defined in PLATFORM.DLL

bblockTableSize
number of blocks described in blockTable

dwBlockTable
Flash parts are programmed one contiguous block at a time. Each block to be programmed must have a corresponding block descriptor.

bpartTypesSize
number of flash parts to add

dwPartTypes
Optional table of supported flash parts. Each entry in the table has the following format:

```
typedef struct
{
        BYTE  cMfgID;              // Manufacturer ID
        BYTE  cPartID;             // Part ID
        WORD  wFlashType;          // Flash algorithm type
        char  szPartName[28];      // Optional description
} DEVICETABLE;
```

The device table is terminated by a descriptor with cMfgID, cPartID and wFlashType set to zero.

Many platforms allow the same BIOS.ROM image to be used with several different flash parts. This table is used when a new part becomes available, the part is not among the parts currently supported by PHLASHNT.EXE and the part uses the same flashing algorithms as one of the supported parts.

procEnable	enable flashing proc
procDisable	disable flashing proc
procBegin	begin flashing proc
procEnd	end flashing proc
procGetBlock	get next BIOS block for backup proc
procCmdLine	process custom command line options proc
procSense	custom autosense proc
procIsFlashable	custom OEM proc to determine if ok to proceed
procReboot	custom reboot proc
procCheckSum	custom checksum proc to be used if the BIOS ROM image checksum is not zero

A sample source code for PLATFORM.DLL file is shown in Appendix B3.

3.3 Block Table Format

Block table consists of a list of block descriptors. Each block descriptor in the block table is defined by the following structure:

```
typedef struct
      {
      DWORD dwBlockSize;        // Number of bytes in the block
      DWORD dwFileOffset;       // Offset within BIOS.ROM file
      DWORD dwLinearAddress;    // Linear 32-bit address of flash ROM
      BYTE  cMfgID;             // Manufacturer ID or zero
      BYTE  cPartID;            // Part ID or zero
      WORD  wBlockAttr;         // Block attributes
      } BLOCK_DESCRIPTOR;
```

The block table is terminated by a descriptor with all entries set to zero.

dwBlockSize	Block Size in bytes. Blocks must be contiguous.
dwFileOffset	Offset of this block within the BIOS.ROM file.
dwLinearAddress	Starting address of this block in 32-bit address space
cMfgID	Manufacturer ID or zero to auto-sense.
cPartID	Part ID or zero to auto-sense.
wBlockAttr	Determines actions to be taken for this block. Must be a combination of the following flags:

ATTR_ZERO	Block must be zeroed before programming
ATTR_ERASE	Block must be erased before programming
ATTR_SAVE	Save content of this block before prog.
ATTR_PROG	Program this block
ATTR_RESTORE	Restore content of this block after prog.

Only one of ATTR_SAVE, ATTR_PROG, ATTR_RESTORE can be used at a time. If no attribute is specified, then PHLASHNT.EXE leaves the block unchanged. However, even if all of these are omitted, procedures BeginFlash() and EndFlash() are still called. BeginFlash() and EndFlash() can be used when two blocks are in different flash devices, or when a boot block requires additional functions to enable the block for write and to disable it before next block is programmed. BeginFlash() can also be used for conditional block processing. If BeginFlash() returns nonzero, the current block is not processed.

Each ATTR_SAVE block must be followed by an ATTR_RESTORE block before another ATTR_SAVE block can be used.

Note that for a given flash memory range there may be several block descriptors. For example to preserve a 16k flash memory Boot Block in 64k erasable flash memory block, three block descriptors would be used. First descriptor to save the 16k boot block, second to erase and program 64k and third to restore the boot block.

To reduce the time required for flashing, it is recommended that the ATTR_ZERO flag is not used, because this will avoid the zeroing step and cut the flashing time in half. Only few of the older flash memory types suggested that part is zeroed before it is re-programmed. Most parts do not require this operation.

3.3.3 Multiple Flash Blocks

The block table will be used to support multiple device flashing and multiple blocks within each device. For such platforms the ROM image file must contain the images for all flash parts to be programmed and the Block Table must contain proper offsets and lengths for each block of data to be flashed.

To properly configure the platform before and after each flash block, PHLASHNT.EXE will call function BeginFlash(Block_Index) to allow PLATFORM.DLL to perform any such set-up. It is the responsibility of BeginFlash() & EndFlash() functions to perform any initialization or termination between blocks as needed.

3.3.4 Processing Boot Blocks and ESCD storage

To program a memory range in the flash memory, there may have to be several different block descriptors, for the same memory range. This may be needed to preserve boot block or ESCD storage within a single 'erase' memory range.

Many flash parts are erased by a small number of write operations, one for each memory block. For example Intel 28F400 flash memory has seven blocks (one 16k boot block, two 8k parameter blocks, one 64k main block and three 128k extended blocks). This part can be erased with only seven write operations.

For other parts, erase function may erase 64k of flash memory at a time, regardless of division of this range into boot and parameter blocks. In such cases it is important that there are three block descriptors for such a 64k range of memory. The first block descriptor in the table is used to save boot block, second block descriptor to erase and program the parameter blocks and the third descriptor to restore the boot block in this range.

Some parts require that a additional platform dependent actions need to be taken before a boot block can be programmed. For example Intel parts require that VHH voltage, in addition to VPP voltage, is properly set. In such cases block descriptor must also have such a functions in BeginFlash() and EndFlash() procedures (called before and after each block).

3.3.5 Block Table Examples

The code in the following examples would be placed in the "blockTable" section of PLATFORM.CPP.

Erase a 4KB block at FC000.

```
DD      4 * 1024                ; 4KB
DD      0                       ; File offset
DD      000FC000h               ; Linear address of this block
DB      0                       ; Mfg ID  (0 = default)
DB      0                       ; Part ID (0 = default)
DW      ATTR_ERASE              ; Action flags
```

Zero, then program a 128KB block at E0000 on the specified part.

```
DD      128 * 1024              ; 128KB
DD      0                       ; File offset
DD      000E0000h               ; Linear address of this block
DB      01h                     ; Mfg ID  (0 = default)
DB      20h                     ; Part ID (0 = default)
DW      ATTR_ZERO OR ATTR_PROG ; Action flags
```

3.4 PLATFORM.DLL Functions

Currently supported functions are:

The following functions contained in PLATFORM.DLL are accessed by PHLASHNT.EXE to implement platform-dependent functionality:

EnableFlash
DisableFlash
BeginFlash
EndFlash
GetBlock
CmdLine
AutoSense
IsFlashable
Reboot
CheckSum

The following functions allow PHLASHNT.EXE to access global variables and data structures contained in PLATFORM.DLL:

GetBIOSFileSize
GetManufactID
GetPartID
GetFlags
GetImageBuf
GetMfgIDAddr
GetPartIDAddr
GetRetryCount
GetblockTableSize
GetpartTypesSize
GetBlockTable
GetpartTypes
GetDLLVersion
GetROMFileName
GetDLLFuncDefine

3.4.1 Function EnableFlash()

Entry: None
Returns: Error code (or zero)

This function must be present in the PLATFORM.DLL. It is called before any attempt to access the flash memory. Actions typically performed by this function include:

Map flash device into memory
Disable Cache, Shadowing and Power management
Flush Cache
Disable PCI bridge
Enable ROM for write (VPP on)

Most platforms require that a jumper is changed before the part can be enabled for flashing. EnableFlash() procedure must verify that this jumper has been removed and return an error code if it was determined that jumper settings are incorrect. See Appendix B for error codes.

3.4.2 Function DisableFlash()

Entry: None
Returns: Error code (or zero)

This optional function is called after the last block has been programmed (or error was detected). It will be called immediately before PHLASHNT.EXE exits (typically as a last call before re-boot). It should perform all functions necessary to be able to cleanlv re-

Action typically performed by this function include:

Disable ROM Write (VPP off)

3.4.3 Function BeginFlash(DWORD Block_Index)

Entry:	Index of the block about to be programmed (or zero if table not used)
Exit:	None
Returns:	Error code (or zero)

This optional function is called by PHLASHNT.EXE immediately before the flash block is processed. It will be called for each block (found in the block table).

Actions typically performed by this function include:

Save BOOT block before it is erased
Switch from one device to another (on platforms with multiple devices)
Enable VHH for boot block re-program
Determine if the current block should be processed

If BeginFlash() returns nonzero, the current block will not be processed.

3.4.4 Function EndFlash(DWORD Block_Index)

Entry:	Index of the block just programmed (or zero of table not used)
Returns:	Error code (or zero)

This optional function is called by PHLASHNT.EXE immediately after the flash block was processed. It will be called for each block (found in the block table).

Actions typically performed by this function include:

Restore BOOT block saved by BeginFlash() function
Clean-up between programming two different devices
Disable VHH if boot block was just programmed

3.4.5 Function GetBlock(DWORD Index, DWORD Buffer_Address)

Entry:	Index of the block to be copied and linear address of a 64k buffer
Exit:	Buffer is filled with the next block of existing BIOS ROM image
Returns:	Negative error code, zero, or positive block index

This optional function is called by PHLASHNT.EXE whenever the /BACKUP flag is specified. GetBlock() is used to assist when saving the existing content of the flash memory before the flash memory is changed. Because many BIOS images are de-compressed into shadow RAM, it is not always possible for PHLASHNT.EXE to access all of the BIOS ROM image without platform dependent system setup. Function GetBlock() is necessary to allow for platform dependent accessing of the existing BIOS ROM image. BIOS ROM image is saved by PHLASHNT.EXE using the following steps:

1) Call GetBlock(Index, Buffer) with Index set to 0 and the 64k buffer, pointed to by the parameter Buffer, filled with a pre-defined pattern. If the pattern in the buffer is not changed, program exits with error. If the pattern is changed, then the buffer is saved as the first 64k block in the BIOS.BAK file, then program proceed to next step.

2) Call GetBlock(Index, Buffer) with Index set to the value returned by the previous call to GetBlock(), save the 64k buffer into BIOS.BAK and repeat this until the value returned by GetBlock() is a non-positive number.

3) If the last value returned by GetBlock() is zero, then proceed with flashing of the memory. If the last value returned is negative error code, report the error, delete BIOS.BAK and exit.

It is the responsibility of the GetBlock() implementation to ensure that the platform is in a proper state to allow the GetBlock() to copy BIOS ROM image into the buffer and that the platform is restored to the original mode before GetBlock() returns control to PHLASHNT.EXE. In particular, GetBlock() is called before a call is made to EnableFlash(). The buffer pointer passed to GetBlock() is always in the real memory range below 640k, to allow direct transfer to disk.

3.4.6 Function CmdLine(char far *szOptions)

Entry:	Pointer to a string with the platform specific command line options
Returns:	Error code (or zero)

This optional function is called by PHLASHNT.EXE immediately after the PLATFORM.DLL was read in. It is passed the address of the string containing all the platform specific command line parameters (specified after the equal sign with /PLATFORM="..." command line option). The string includes the leading and trailing double quotes, if any.

3.4.7 Function AutoSense()

Entry:	Manufacturer and device IDs retrieved from PLATFORM.INI header
Returns:	New ID retrieved from the flash part (or zero)

This function is called by PHLASHNT.EXE immediately after EnableFlash() function in the PLATFORM.DLL was called. The AutoSense() function enables auto detection of memory flash parts when "non-standard" memory organization is used for the flash memory. For example when two separate parts are used for even and odd BIOS addresses (in which case the conventional auto detect mechanism will fail), this function can be used to obtain and verify ID for each of the parts.

IDs are one byte long and are packed into a DWORD, with manufacturer ID in BYTE 0 and device ID in BYTE 1.

3.4.8 Function IsFlashable(char far *szErrorMsg)

Entry: Pointer to string to contain returned error message.
Exit: szErrorMsg containing error message string.
Returns: Error code (or zero)

This optional function is called before EnableFlash() to determine if it is ok to proceed. If the function returns a nonzero error code, the string szErrorMsg is displayed and the program terminates. Up to 254 bytes plus a terminating NULL may be returned in szErrorMsg.

An example of how this might be used is: for the same platform, an OEM sells a system with and without Plug and Play capabilities. The IsFlashable() function can determine if the system currently has Plug and Play and will not flash a Plug and Play BIOS on a platform that doesn't already have it.

3.4.9 Function Reboot()

Entry: None.
Returns: None.

This optional function is called after programming is complete to reset the system. If provided, this is called instead of PHLASHNT's own reboot code.

3.4.10 Function CheckSum()

Entry: None.
Returns: Error code (or zero)

This optional function is called before programming to determine if the checksum of the BIOS ROM image is correct. Normally, the BIOS ROM image checksum for a NuBIOS image is zero. This routine may be used to provide an alternative checksum verification method when it is known that the ROM image checksum will not be zero. If this function returns non-zero, PHLASHNT will terminate.

3.4.11 Function GetBIOSFileSize()

Entry: None.
Returns: Size of BIOS image

This function returns the contents of the global variable dwFileSize from PLATFORM.DLL.

3.4.12 Function GetManufactID()

Entry: None.
Returns: Manufacturer ID

This function returns the contents of the global variable bManufactID from PLATFORM.DLL.

3.4.13 Function GetPartID()

Entry: None.
Returns: Part ID

This function returns the contents of the global variable bPartID from PLATFORM.DLL.

3.4.14 Function GetFlags()

Entry: None.
Returns: option flags

This function returns the contents of the global variable dwFlags from PLATFORM.DLL.

3.4.15 Function GetImageBuf()

Entry: None.
Returns: Location of Image Bufffer

This function returns the contents of the global variable dwImageBuf from PLATFORM.DLL.

3.4.16 Function GetMfgIDAddr()

Entry: None.
Returns: Address where Manufacturer ID is located

This function returns the contents of the global variable dwMfgIDAddr from PLATFORM.DLL.

3.4.17 Function GetPartIDAddr()

Entry: None.
Returns: Address where Part ID is located

This function returns the contents of the global variable dwPartIDAddr from PLATFORM.DLL.

3.4.18 Function GetRetryCount()

Entry: None.
Returns: Number of times to retry flashing if failure occurs

This function returns the contents of the global variable bRetryCount from PLATFORM.DLL.

3.4.19 Function GetblockTableSize()

Entry: None.
Returns: Number of blocks in blockTable

This function returns the contents of the global variable bblockTableSize from PLATFORM.DLL.

3.4.20 Function GetpartTypesSize()

Entry: None.
Returns: Number of additional flash parts that PLATFORM.DLL will describe

This function returns the contents of the global variable bpartTypesSize from PLATFORM.DLL.

3.4.21 Function GetBlockTable()

Entry: None.
Returns: address of blockTable

This function returns the address of the global structure blockTable from PLATFORM.DLL.

3.4.22 Function GetpartTypes()

Entry: None.
Returns: address of partTypes

This function returns the address of the global structure partTypes from PLATFORM.DLL.

3.4.23 Function GetDLLVersion()

Entry: None.
Returns: Version of PLATFORM.DLL

This function returns the contents of the global variable szVersion from PLATFORM.DLL.

3.4.24 Function GetROMFileName()

Entry: None.
Returns: Name of BIOS ROM file

This function returns the contents of the global variable szROMFileName from PLATFORM.DLL.

3.4.25 Function GetDLLFuncDefine()

Entry: None.
Returns: Indicates what platform-dependent functions are defined in PLATFORM.DLL

This function returns the contents of the global variable dwDLLFuncDefine from PLATFORM.DLL.

4.0 BIOS.ROM Detail

The ROM image file size must be large enough to contain all blocks in the flash device(s) to be programmed. Any required post-processing of the BIOS image, such as boot block swapping or data compression, must be already incorporated into the ROM image file.

5.0 General Implementation Guidelines

Programs will be developed using Microsoft Visual C++, V4.2 or later.

Because it is expected that this program will evolve as new devices become available and because code size is not critical, the style of coding should be such that code readability and clarity should have higher priority than executable code compactness.

However, because the program may also be used in production environments, it should be structured as to allow shortest possible time for part flashing. In particular, it should have a mode where non-critical user interface notifications can be disabled; and whenever possible, time consuming flashing functions should be written in optimized assembly language.

AppendixB1-Future enhancements

Completion Codes with Keyboard LEDs

If the program fails to complete any of the three major stages of the flashing process, program will use keyboard LEDs to inform user at which stage the program failed. At the start of the program CAPS_LOCK, NUM_LOCK and SCROLL_LOCK LEDs on the keyboard will be turned on. The failure at each of the three stages is indicated as follows:

Keyboard LEDs ON	Description
CAPS, NUM, SCROLL	Before reading platform.dll
CAPS, NUM	Before platform init
NUM	After platform init
none	Successful completion

Appendix B2-PHLASHNT.H Completion and error codes

FFh	Memory alloc for BIOS.BAK buffer failed
FEh	BIOS.BAK already exists (rename or delete it)
FDh	File Create failed on BIOS.BAK
FCh	File Write failed on BIOS.BAK
FBh	File Close failed on BIOS.BAK
FAh	BIOS backup not supported in PLATFORM.DLL
F9h	File Open failed on PLATFORM.DLL
F8h	File Read failed on PLATFORM.DLL
F7h	File Close failed on PLATFORM.DLL
F6h	Failed to locate signature bytes in PLATFORM.DLL
F5h	Unsupported PLATFORM.DLL file version
F2h	Device table in PLATFORM.DLL has too many entries
F1h	Device table in PLATFORM.DLL has unsupported flash type
F0h	Combined SAVE or RESTORE attributes in PLATFORM.DLL
EFh	SAVE block without matching RESTORE block in PLATFORM.DLL
ECh	Part ID not found in table of supported parts
EBh	PLATFORM.DLL found errors in command line parameters
EAh	Alloc for BIOS ROM image failed
E9h	Open failed on BIOS ROM image file
E8h	Read failed on BIOS ROM image file
E6h	File Close failed on BIOS.ROM
E4h	Attempt to read flash memory ID failed
E3h	PLATFORM.DLL failed to return flash memory ID
E2h	Could not find BCPSYS block in BIOS ROM file image
E1h	File does not contain the same BIOS part number
E0h	File does not contain different version of BIOS ROM image
DFh	Data written to flash does not match BIOS ROM image
DEh	Write to flash memory failed
DDh	Erase flash memory block failed
DCh	VPP is not at expected level
DBh	Erase sequence failed
DAh	New DMI string is too large
D9h	The BIOS ROM file may not be used with this system
D8h	Alloc for DMI OEM string failed
D7h	No space for the specified DMI OEM string in the BIOS ROM
D6h	DMI OEM strings not supported (BCPDMI 0.1+ required)
D5h	Could not find BCPDMI block in BIOS ROM file image
D3h	BIOS ROM file may be corrupt (checksum not zero)
D2h	BIOS ROM file size doesn't match flash part size
D1h	DMI system and chassis strings require BCPDMI 2.1+

Appendix B3-PLATFORM.DLL Sample source code

PLATFORM.CPP

```
/*
 *
 * TITLE   PLATFORM.CPP   - 32-Bit DLL to provide platform dependent functionality
 *                          to PHLASHNT.EXE
 *
 * Copyright (c) 1997 by Phoenix Technologies, Ltd., All Rights Reserved.
 * Phoenix Technologies Ltd. CONFIDENTIAL.
 *
 ***********************************************************************
 *
 * Filename:          PLATFORM.CPP
 *
 * Project:           PHLASHNT.EXE
 *
 * Functional Block:
 *
 * Comments:    The variable dwDLLFuncDefine is used in PHLASHNT to determine
 *              what platform specific functions in PLATFORM.DLL have been
 *              defined.  Listed below are the possible values of
 *              dwDLLFuncDefine and what functions they indicate are defined
 *              in this module:
 *
 *                      DLL_ENABLEFLASH        EnableFlash
 *                      DLL_DISABLEFLASH       DisableFlash
 *                      DLL_BEGINFLASH         BeginFlash
 *                      DLL_ENDFLASH           EndFlash
 *                      DLL_GETBLOCK           GetBlock
 *                      DLL_CMDLINE            CmdLine
 *                      DLL_AUTOSENSE          AutoSense
 *                      DLL_ISFLASHABLE        IsFlashable
 *                      DLL_REBOOT             Reboot
 *                      DLL_CHECKSUM           CheckSum
 *
 * Contents:
 *
 ***********************************************************************
 *
 * Version Control Information:
 *
 * $Log:   K:/nb/archive/nutools/phlash.nt/stage2/drivers/platform.cpv  $
 *
 *         Rev 1.0
 *      Initial revision.
 *
 ***********************************************************************/

#include "stdafx.h"
#include <afxdllx.h>

#include <stdio.h>
#include "D:\NUTOOLS\PHLASH.NT\STAGE2\phlashnt.h"

#define PLATFORM_CPP

        // ===================================
        // Global Variable Declarations
        // ===================================

DWORD   dwFileSize      = 0x00040000;  // ROM image file size
BYTE    bManufactID     = 0x89;        // Manufacturer of flash device
BYTE    bPartID         = 0xBD;        // Part ID of flash device

                                       // Option flags
DWORD   dwFlags = FLAG_BIOSPARTNUM | FLAG_CHECKSUM | FLAG_IMAGESIZE;
DWORD   dwImageBuf      = 0x00200000;  // Linear address of image buffer
DWORD   dwMfgIDAddr     = 0xFFFE0000;  // Linear address of mfg ID
```

```
DWORD   dwPartIDAddr     = 0xFFFE0001;  // Linear address of part ID
BYTE    bRetryCount      = 0;           // Count for /Rn option (default = 0)
char    szVersion[]      = "NT 1.00";   // PLATFORM.DLL version
char    szROMFileName[]  = "BIOS.ROM";  // Name of BIOS image file

                // Indicates what functions are defined in PLATFORM.DLL
DWORD   dwDLLFuncDefine =     DLL_ENABLEFLASH       |
                              DLL_DISABLEFLASH      |
                              DLL_BEGINFLASH        |
                              DLL_ENDFLASH          |
                              DLL_GETBLOCK          |
                              DLL_CMDLINE           |
                              DLL_AUTOSENSE         |
                              DLL_ISFLASHABLE       |
                              DLL_REBOOT            |
                              DLL_CHECKSUM ;

                              // ==========================
                              // define blockTable
                              // ==========================
BYTE    bblockTableSize       = 5;          // number of blocks in blockTable
BLOCK_DESCRIPTOR blockTable[] =
{
        {
        128 * 1024,       // 128KB (Low)
        0,                // File offset
        0xFFFE0000,       // Linear address of flash ROM
        0,                // MfgID  (same as default)
        0,                // PartID (same as default)
        ATTR_ZERO
        },

        {
        128 * 1024,       // 128KB (High)
        0x00020000,       // File offset
        0xFFFE0000,       // Linear address of flash ROM
        0,                // MfgID  (same as default)
        0,                // PartID (same as default)
        ATTR_ZERO
        },

        {
        128 * 1024,       // 128KB (Low)
        0,                // File offset
        0xFFFE0000,       // Linear address of flash ROM
        0,                // MfgID  (same as default)
        0,                // PartID (same as default)
        ATTR_ERASE
        },

        {
        128 * 1024,       // 128KB (Low)
        0,                // File offset
        0xFFFE0000,       // Linear address of flash ROM
        0,                // MfgID  (same as default)
        0,                // PartID (same as default)
        ATTR_PROG
        },

        {
        128 * 1024,       // 128KB (High)
        0x00020000,       // File offset
        0xFFFE0000,       // Linear address of flash ROM
        0,                // MfgID  (same as default)
        0,                // PartID (same as default)
        ATTR_PROG
        },

        {                 // Need to terminate blockTable with
        0,                // a block set to 0.
        0,
        0,
        0,
        0,
```

```
        0
        }
};
                                // ===========================
                                // define partTypes
                                // ===========================
BYTE    bpartTypesSize = 1;          // number of flash parts on platform
                                     // Set to 0 if partTypes is not used!
DEVICETABLE partTypes[] =
{
        {
        0x89,                   // Manufacturer ID of flash device
        0xBD,                   // Device ID of flash device
        1,                      // Flashing algorithm type
        0,                      // IGNORE wPartSize
        "Intel 28F020"          // Name of flash device
        },

        {                       // Need to terminate partTypes with
        0,                      // a block set to 0.
        0,
        0,
        0,
        ""
        }
};



HINSTANCE     hKernel32 = 0;    // NOTE:  Do NOT change this line of code.

 /**********************************************************************
 *                      Functions
 **********************************************************************/

 /**********************************************************************
 *
 * Function Name: DllMain
 *
 * Description:
 * Parameters:
 * Returns:
 *
 * Notes:        DO NOT MODIFY THIS FUNCTION
 *
 **********************************************************************/
extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
        if (dwReason == DLL_PROCESS_ATTACH)
        {
                TRACE0("PLATFORM.DLL Begin!\n");
                hKernel32 = LoadLibrary( "Kernel32.Dll" ); // Get Handle to Kernel32.Dll
        }
        else if (dwReason == DLL_PROCESS_DETACH)
        {
                TRACE0("PLATFORM.DLL Terminating!\n");
                if (hKernel32)
                        FreeLibrary( hKernel32 );
        }
        return 1;    // ok

} // DllMain()

 /**********************************************************************
 *
 * Function Name: EnableFlash
 *
 * Description:
 * Parameters:
```

```
 * Returns:
 *
 * Notes:
 *
 ************************************************************************/
DLL_FUNC_TYPE void EnableFlash()
{

  printf("EnableFlash\n");  //  (stub)

}
/************************************************************************
 *
 * Function Name: DisableFlash
 *
 * Description:
 * Parameters:
 * Returns:
 *
 * Notes:
 *
 ************************************************************************/
DLL_FUNC_TYPE void DisableFlash()
{

  printf("DisableFlash\n");  //  (stub)

}
/************************************************************************
 *
 * Function Name: BeginFlash
 *
 * Description:
 * Parameters:
 * Returns:
 *
 * Notes:
 *
 ************************************************************************/
DLL_FUNC_TYPE short BeginFlash( USHORT blockNumber )
{

  return 0;     // (stub)

}

/************************************************************************
 *
 * Function Name: EndFlash
 *
 * Description:
 * Parameters:
 * Returns:
 *
 * Notes:
 *
 ************************************************************************/
DLL_FUNC_TYPE short EndFlash( USHORT blockNumber )
{

  return 0;     //  (stub)

}
/************************************************************************
 *
 * Function Name: GetBlock
 *
 * Description:
 * Parameters:
 * Returns:
 *
 * Notes:
 *
 ************************************************************************/
DLL_FUNC_TYPE SHORT GetBlock( ULONG dwIndex, PULONG dwDst )
{
```

```
    printf("GetBlock:  dwIndex,dwDst passed in=%d,%d\n", dwIndex, dwDst );  //  (stub)
    return 0;

}
 /*************************************************************************
 *
 * Function Name: CmdLine
 *
 * Description:
 * Parameters:
 * Returns:
 *
 * Notes:
 *
 *************************************************************************/
DLL_FUNC_TYPE USHORT CmdLine( char *PlatformString )
{
    printf("CmdLine\n");  //  (stub)
    return( 0 );         //  (stub)

}
 /*************************************************************************
 *
 * Function Name: AutoSense
 *
 * Description:
 * Parameters:
 * Returns:
 *
 * Notes:
 *               The manufacturer ID is in BYTE 0 and device ID in BYTE 1
 *
 *************************************************************************/
DLL_FUNC_TYPE DWORD AutoSense( DWORD PartMfgID )
{
    return( 0x0000 );    //  (stub)

}
 /*************************************************************************
 *
 * Function Name: IsFlashable
 *
 * Description:
 * Parameters:
 * Returns:
 *
 * Notes:
 *
 *************************************************************************/
DLL_FUNC_TYPE USHORT IsFlashable()
{

    printf("IsFlashable\n" );  //  (stub)
    return 0;

}
 /*************************************************************************
 *
 * Function Name: Reboot
 *
 * Description:
 * Parameters:
 * Returns:
 *
 * Notes:
 *
 *************************************************************************/
DLL_FUNC_TYPE void Reboot()
{

    printf("Reboot\n");  //  (stub)

}
 /*************************************************************************
```

```
 *
 * Function Name: CheckSum
 *
 * Description:
 * Parameters:
 * Returns:
 *
 * Notes:
 *
 ***************************************************************************/
DLL_FUNC_TYPE void CheckSum()
{

  printf("CheckSum\n");  //  (stub)

}


// ===========================================================================
//                NOTE:  DO NOT MODIFY THE FUNCTIONS LISTED BELOW.
// ===========================================================================

DLL_FUNC_TYPE DWORD    GetBIOSFileSize(void)    { return(dwFileSize); }
DLL_FUNC_TYPE BYTE       GetManufactID(void)     { return(bManufactID); }
DLL_FUNC_TYPE BYTE       GetPartID(void)         { return(bPartID); }
DLL_FUNC_TYPE DWORD      GetFlags(void)          { return(dwFlags); }
DLL_FUNC_TYPE DWORD      GetImageBuf(void)       { return(dwImageBuf); }
DLL_FUNC_TYPE DWORD      GetMfgIDAddr(void)      { return(dwMfgIDAddr); }
DLL_FUNC_TYPE DWORD      GetPartIDAddr(void)     { return(dwPartIDAddr); }
DLL_FUNC_TYPE BYTE       GetRetryCount(void)     { return(bRetryCount); }
DLL_FUNC_TYPE BYTE       GetblockTableSize(void) { return(bblockTableSize); }
DLL_FUNC_TYPE BYTE       GetpartTypesSize(void)  { return(bpartTypesSize); }
DLL_FUNC_TYPE BLOCK_DESCRIPTOR  *GetBlockTable(void)  { return(&blockTable[0]); }
DLL_FUNC_TYPE DEVICETABLE        *GetpartTypes(void)  { return(&partTypes[0]); }
DLL_FUNC_TYPE char     *GetDLLVersion(void)    { return(szVersion); }
DLL_FUNC_TYPE char     *GetROMFileName(void)  { return(szROMFileName); }
DLL_FUNC_TYPE DWORD    GetDLLFuncDefine(void) { return(dwDLLFuncDefine); }
```

Appendix C

Abstract:

This paper presents a proposal for a new BIOS service, which will be known as the **BIOS32 Service Directory.** The new service will provide information about those services in the BIOS that are designed for callers running in a 32-bit code segment. (The BIOS32 Service Directory will itself be a 32-bit BIOS service.) The expected clients of this service are 32-bit operating systems and 32-bit device drivers. The expected providers of this service are BIOS vendors that implement one or more 32-bit BIOS services.

1 Introduction

The BIOS32 Service Directory proposal came into being during the attempts to establish a 32-bit code interface for the Peripheral Component Interconnect (PCI) standard. A major problem that needed solving was: How does a 32-bit caller determine the existence of a 32-bit BIOS service? The drawbacks of a functional interface (i.e., an entry point that control is transferred to) are clear in the case where the functional interface is non-extant. A recoverable method would seem to entail a search for a signature as proof that a functional interface does exist. In fact, one of the first BIOS 32-bit services, EISA, uses an existence signature at a fixed location in the 0F000h segment. However, as more 32-bit BIOS services are required and come into existence, it is obvious that providing a signature (and any associated information, such as entry points, segment requirements, etc.) for each is a costly usage of a scarce memory resource. Hence, the idea behind the

The BIOS32 Service Directory uses a single signature to indicate the existence of a generic 32-bit service that returns information on all specific 32-bit services

A further justification for implementing a general solution versus continuing to solve the problem on a case-by-case basis is seen in the obvious benefits a standard provides to the industry. Code reusability, modular ("plug-in"-able) implementation of new 32-bit BIOS interfaces, thorough specification of calling environment requirements, etc., are a few benefits which come to mind.

The BIOS32 Service Directory has two components: the Header and the Calling Interface. The Header is a static data structure that includes the signature and which is located in a well-known memory range. The existence of a valid Header implies the existence of the Calling Interface and, in fact, describes its entry point. The Calling Interface is a body of code and data which exists in a separate memory space apart from the Header and any particular 32-bit BIOS services. It provides a functional interface through which a caller receives information about a particular 32-bit BIOS service. (Section 0 describes a hypothetical memory map of the Header, Calling Interface, and the "XYZ" 32-bit BIOS service.)

The design of the Calling Interface is extensible in two dimensions. First, it is a function-based interface - future revisions of the service can incorporate new features, as they become necessary. Second, specific 32-bit BIOS services will be represented by a 4-byte Component ID. This will enable both OS callers and BIOS32 Service Directory implementors alike to easily add new 32-bit BIOS services as they become available.

Interface to the BIOS32 Service Directory, and a summary with example

2 The BIOS32 Service Directory Header

A BIOS which implements the BIOS32 Service Directory must embed a specific, contiguous 16-byte pattern somewhere in the physical address range 0E0000h - 0FFFFFh. The pattern must be paragraph aligned (i.e., it must start on a 16-byte boundary). This pattern is known as the BIOS32 Service Directory Header.

The Header is comprised of six distinct fields. The following table describes each field.

Offset	Size	Description
0	4 bytes	The ASCII signature "_32_". This string is packed left to right: offset 0 is '_' (underscore), offset 1 is '3', offset 2 is '2', offset 3 is another '_'.
4	4 bytes	The entry point for the BIOS32 Service Directory Calling Interface. This is a 32-bit linear (i.e., not segment:offset) physical address.
8	1 byte	The revision level of the BIOS32 Service Directory Header and Calling Interface. The current revision is 0h.
9	1 byte	The length of the BIOS32 Service Directory Header. This is measured in units of paragraphs (16 bytes). The current Header has a length of 1h.
10	1 byte	The BIOS32 Service Directory Header checksum. This is a value which makes the cumulative ADD value of all bytes in the Header equal to 0h.
11	5 bytes	This field is reserved and should be set to 0h.

TABLE 1: The BIOS32 Service Directory Header

Clients of the BIOS32 Service Directory should first determine its existence by locating the Header. This is done by scanning 0E0000h to 0FFFF0h in paragraph increments and looking for a signature match ("_32_") in the first 4 bytes of each paragraph. When, and if, the signature is detected the client should perform a checksum of all bytes in the Header. (The Header length, in paragraphs, is found at offset 9h.) All bytes in the Header should ADD together with a result of 0h. If the checksum is valid then the 32-bit entry point field can be used as the address for the BIOS32 Service Directory Calling Interface. If the Header is not found then the BIOS32 Service Directory does not exist on the platform.

3 The BIOS32 Service Directory Calling Interface

If the BIOS32 Service Directory existence has been determined (via the Header test, above) then the 32-bit physical address found in the Header can be used as the entry point to the Calling Interface. Clients should CALL FAR to this address. The client's calling environment has the following requirements.

3.1 Code Segment

The CS code segment selector must be set up with a segment descriptor with the following values:

- The base address must be less than or equal to the (4kb) page address of the page that contains the entry point. For example, if the entry point is 0FFF81234h then the base address must be less than or equal to 0FFF81000h.
- The limit must be such that the base address plus the limit generate an address that is greater than or equal to the last address on the (4kb) page which follows the page containing the entry point. For example, if the entry point is 0FFF81234h then the base address plus the limit must be greater than or equal to 0FFF82FFFh.
- Simply stated, the base address and the limit must "encompass" both the page that contains the entry point and the following page.
- The segment type must be 100b (code, execute only) or 101b (code, execute/read). However, the implementors of the Service Directory cannot assume read access to the CS code segment.
- The system bit must be 1 (nonsystem segment).
- It is recommended that the Descriptor Privilege Level (DPL) be 0. (The CS descriptor DPL becomes the Current Privilege Level, or CPL). If the CPL is not 0, then the OS must provide trapping and virtualization services for ring 0 privileged instructions (such as those that access CRx). Note also the dependency of this field on the IOPL field in EFLAGS (see Section 0).
- The Default Size bit must be 1 (32 bits).

The BIOS32 Service Directory entry point, and its associated code and data, may be located anywhere within the 4Gb physical address space. However, it is guaranteed to be physically contiguous (i.e., it will be delivered in ROM or FLASH space) and to fit within two pages (i.e., it will not span three pages).

3.2 Data Segments

The DS data segment selector must be set up with a segment descriptor with the following values:

- The base address must be equal to the CS base address.
- The limit must be greater than or equal to the CS limit.
- The segment type must be 000b (data, read only) or 001 (data, read/write). However, the implementors of the Service Directory cannot assume write access to the DS data segment.
- The system bit must be 1 (nonsystem segment).
- The Descriptor Privilege Level (DPL) must be greater than or equal to CPL (see the DPL field in Section 0).

There are no requirements concerning the ES, FS, and GS data segment selectors.

3.3 Stack Segment

The SS stack segment selector must be set up with a segment descriptor with the following values:

- The segment type must be 011b (data, read/write, expand-down) or 001b data, read/write, expand-up).
- The system bit must be 1 (nonsystem segment).
- The Descriptor Privilege Level (DPL) must be equal to the CPL (see the DPL field in Section 0).
- The Default Size bit must be 1 (32 bits).
- The Granularity bit must be 1 (4Kb).

Note that the above settings ensure a stack size of at least 4kb. It is the caller's responsibility to ensure that there is at least 1kb of unused stack available.

3.4 Paging

Paging may or may not be enabled. If paging is enabled, then the address space that is described by the CS and DS selectors must be linearly contiguous. That is, the original physical contiguity of the Calling Interface as found in ROM or FLASH must be preserved. (The Calling Interface code and data is written to be position-independent and EIP-relative).

3.5 IOPL

In order for the Calling Interface to execute I/O instructions, the I/O Privilege Level (IOPL) field in EFLAGS must be greater than or equal to the CPL (see the DPL field in Section 0).

The BIOS32 Service Directory Calling Interface is function-based and all parameters are passed in registers. If a register is not specified as an output parameter for a function, then it will be preserved. All flags are preserved. Function values are passed as input parameters in register BL. Return status is passed back in register AL. A return status of 00h indicates that the function was successful. A return status of 80h indicates that the requested function (in BL) is not supported. Other AL return values are defined by the individual functions. There is currently one BIOS32 Service Directory function defined. It is specified below.

3.6 The Component Existence Function (BL = 0h)

The Component Existence function returns information about whether a specific 32-bit BIOS service exists and, if it does, what memory space it occupies.

Input:

BL, 0h

EAX, Component ID

The Component ID is a 4-byte ASCII string which uniquely identifies the 32-bit BIOS service. The specifications for particular BIOS services define their own Component IDs. (It is important that those specifications define whether the Component ID string is packed left to right, or right to left.)

EBX, Reserved, set to 0h

Output:

If requested 32-bit service does not exist

AL = 81h

If requested 32-bit service does exist

AL = 0h

EBX = base address of 32-bit BIOS service

ECX = length of 32-bit BIOS service

EDX = offset (from EBX) of 32-bit BIOS service entry point

The meaning of the EBX, ECX, and EDX registers is dependent on the particular 32-bit BIOS service specification. That is, they may represent exact values for setting up segment selectors, minimal "encompassing" values, etc.

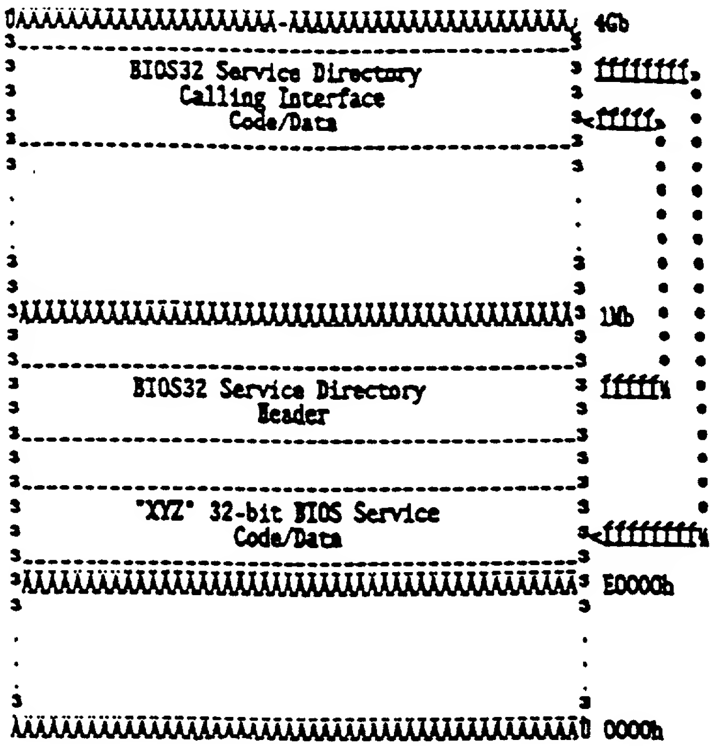
3.7 Future Functions

Future BIOS32 Service Directory functions may be defined in subsequent revisions of this document. The function parameter interface is not constrained to register passing and may employ input/output parameters on the stack. This is feasible due to the static definition of the stack (see Section 0).

4 Summary

This paper describes the BIOS32 Service Directory. It identifies two distinct memory components: the Header and the Calling Interface. The Header is a paragraph of signature data located in the memory range 0E0000h - 0FFFFFh. The Calling Interface is a body of code that occupies a physically contiguous section of memory anywhere in the 4Gb address space and is less than two pages (8kb) in length. The Header contains a pointer into the Calling Interface memory area.

The BIOS 32 Service Directory Calling Interface functions describe additional memory areas that contain code and data for specific 32-bit BIOS services. A memory map for a hypothetical Header, Calling Interface and the "XYZ" 32-bit BIOS service follows.



This paper provides for a standard method of accessing 32-bit BIOS services. The extensibility of the BIOS32 Service Directory Calling Interface may be used easily to advance a standard method as new 32-bit BIOS services are recognized and implemented.

```
#ifdef TESTEXEC

 //  test bios execute function.

 printf ("phadtest - test bios execution breakpoint\n");

 ioExec = (PIOC_EXEC1)systemBuffer;
 ioExec->biosFunction = biosShadowAreaBaseVA + biosServiceDirectoryVA;
 strncpy ((PUCHAR)(&ioExec->regEAX), BIOS_PM_SIGNATURE, 4);
 ioExec->regEBX = 0L;

 if (DeviceIoControl(hDriver,
      IOCTL_BIOS_EXEC,
      bufferPtr,
      256,
      bufferPtr,
      256,
      &actualXfr,
      NULL))  {
   printf ("IOCTL 2052 -BIOS Execute (%s) - success\n",
 BIOS_PM_SIGNATURE);
   printf ("# bytes returned: %Ld\n", actualXfr);

   printf ("Register contents:\n");
   printf ("AL  = %X\n",  (ioExec->regEAX ) & 0xff);
   printf ("EBX = %LX\n",  ioExec->regEBX );
   printf ("ECX = %LX\n",  ioExec->regECX );
   printf ("EDX = %LX\n",  ioExec->regEDX );

  }else{

   ioCode = GetLastError();
   printf ("IOCTL 2050 failed on %LX\n", ioCode);

  }

 #endif
```

Appendix D1

```
//-----------------------------------------------------------------
   case IOCTL_BIOS_EXEC:

#ifdef DEBUGMZ
        MzIoKdPrint (4);
#endif

        //  Function 4 = Code 2052 is 'BIOS Execute'.
        //  The function executes code in the shadow area.

        ioExecBios = (PIOC_EXEC1) (Irp->AssociatedIrp.SystemBuffer);


        if (foundBios32 == FALSE) {          // if bios32 not found in.
         ioExecBios->runStat = FALSE;        // didn't run.
         return (IoCtlFinish (Irp, STATUS_SUCCESS, sizeof (IOC_EXEC1)) );
        }

        // check the range of the requested function.

        actualXfrd = rangeCheck (ioExecBios->biosFunction,    // function address.
                                 1L,                          // only look at e.p.
                                 MODE_SHADOW,                 // read/exec mode, BIOS.
                                 lenBiosRom                   // length of rom space.
                                );

         // failed range check.

         if (actualXfrd == 0) {

             return (IoCtlFinish (Irp, STATUS_ACCESS_DENIED, sizeof (IOC_EXEC1)) );

         }


       // now execute the function.
       if ((ioExecBios->biosFunction)== ((ULONG)bios32ServiceEntryPoint+(ULONG)romMapPtr
           connectBios (ioExecBios, ioExecBios->biosFunction);
       else
           execBios (ioExecBios, ioExecBios->biosFunction);

       ioExecBios->runStat = 1;

           return ( IoCtlFinish(Irp, STATUS_SUCCESS, sizeof (IOC_EXEC1)) );
                         // don't forget to return exec struct..

   //-----------------------------------------------------------------
```

Appendix D2

```
//  this function creates a register context and calls the
//  entry point.

void
execBios(
    PIOC_EXEC1      systembuffer,
    ULONG           entrypoint
    )
{

#ifdef BIOS_FAR_CALL

    void    (far *address32)(PIOC_EXEC1);

#else

    void    (*address32)(PIOC_EXEC1);

#endif

    *address32 = (void (*)(PIOC_EXEC1))entrypoint;

 _asm {
    mov eax, [systembuffer]     // ptr to the systembuffer
    push    eax             //
    push    cs              // make it a FAR call
    call    [address32]         // call to the Serive Directory
 }

 //     (*address32)(systembuffer);         // this is service interface.

 }
```

Appendix D3